SERIE
AUTOAPRENDIZAJE
ACELERADO

colección
m/b

ZX Spectrum

código máquina simplificado (volumen-1)

para principiantes

James Walsh

JAMES WALSH

EL CODIGO MAQUINA DEL SPECTRUM SIMPLIFICADO

VOLUMEN UNO

PARA PRINCIPIANTES

EDICIONES TECNICAS REDE, S.A.
Apartado 35.400
BARCELONA

Publicado en Gran Bretaña por
INTERFACE, London
con el título de SPECTRUM MACHINE CODE MADE EASY-Vol. 1—For Beginners

© James Walsh, 1983

Traducción: Ramón Vilas de Escauriaza

Edición Española: © 1984 EDICIONES TECNICAS REDE, S.A.

Todos los derechos quedan reservados. El contenido de este libro no puede ser reproducido, ni total ni parcialmente, ni incorporarse a ningún sistema de archivo de datos reutilizables, ni transmitirse en forma alguna o por cualquier medio electrónico, mecánico o de fotocopia, ni grabarse y tampoco puede utilizarse por procedimiento distinto a los indicados, información contenida en este libro sin el permiso previo del propietario de los derechos del mismo. No se expresan ni se implican garantías con respecto al contenido del libro ni su adecuación para finalidad alguna.

ISBN: 84-247-0203-4

Impreso en España — Printed in Spain

Dep. Legal: B. 40773/84 —REDEPRINT— Barcelona

DEDICATORIA

A Margy, de todos nosotros con amor.
A Charles y Emma: GRACIAS por su gran ayuda.

Al final nos quedaremos mudos.
No podemos remediarlo.
Parte de lo que se ha hecho podemos computarlo.
Pero sin que sepamos qué es lo que ha ocurrido.

Robert Burns

Sumario

INTRODUCCION 11

CAPITULO 1: ¿POR QUE EL CODIGO MAQUINA? 15
La función del intérprete en el ordenador. Mayor velocidad en Código Máquina. El ordenador visto como un interruptor eléctrico. Las «combinaciones» de decisiones, una muestra de las posibilidades del ordenador.

CAPITULO 2: PALABRAS FORMADAS POR NUMEROS. 27
El enlace del ordenador con dispositivos exteriores. Dos tipos de memoria: la RAM y la ROM. Del sistema decimal al binario. El sistema hexadecimal y su utilidad. Los cálculos en aritmética binaria.

CAPITULO 3: EL DIRECCIONAMIENTO DE LAS POSICIONES EN LA MEMORIA DEL SPECTRUM 41
Estructura y organización de la memoria del Spectrum. La memoria considerada como un edificio de viviendas. Determinación de los bytes de memoria que ocupa un programa. La función de «PEEK» y la de «POKE». La ejecución de rutinas en Código Máquina. Cuatro formas distintas de cargar el Código Máquina.

CAPITULO 4: OPERACIONES CON LOS REGISTROS 53
En Código Máquina hay registros en lugar de variables. El lenguaje ensamblador. Los mnemónicos. Las ventajas que ofrece un ensamblador. Un programa demostrativo de lo que es el Código Máquina. La introducción de números en los registros.

CAPITULO 5: COMO REALIZAR SUMAS CON LOS REGISTROS 67
La suma de registros dobles. La carga de un número en un registro doble. En Código Máquina no hay códigos de error. Cuando el Código Máquina exige diez veces menos memoria que el BASIC. La suma de registros sencillos. La suma con acarreo. Como cargar el contenido de una posición de memoria en un registro. Restas o substracciones de números y registros de otros registros. La resta con acarreo. Operaciones con el señalizador de acarreo.

CAPITULO 6: PROGRAMAS ENSAMBLADORES, DESENSAMBLADORES Y CORRECTORES (DEPURADORES). 89
El lenguaje máquina y el lenguaje ensamblador, comparados. El trabajo que realiza un ensamblador. La misión de los desensambladores. Los programas correctores (depuradores). Cómo funciona un programa monitor. Cómo utilizar un ensamblador. Cómo pasar tu programa al ensamblador. Los desensambladores. Cómo interrumpir el Código Máquina. Los monitores. Los programas correctores (depuradores). ¿Qué programa elegir? Cuatro instrucciones muy útiles: INC, DEC, RET y NOP. INC. El incremento de una posición concreta de la memoria. DEC. RET. NOP.

CAPITULO 7: LA CONFECCION DE UN PROGRAMA..... 127
La idea. Elaboración de la estructura. Diagrama de flujo (2.ª versión). Ahora, en el BASIC. ¿Cómo sumamos los números de 16 bits? Traslado de los valores iniciales del BASIC para su uso en Código Máquina. ¿Cómo nos introducimos en el Código Máquina? Estudio de todas las instrucciones del programa examinado. Conversión de instrucciones al lenguaje ensamblador. Ejecución «en seco». Introducción del programa en el ensamblador. Final del programa.

CAPITULO 8: SALTANDO DE UN LUGAR A OTRO 159
El registro PC. La instrucción de salto. Las instrucciones de salto condicional más utilizadas. El salto relativo. Los saltos relativos condicionales.

CAPITULO 9: UTILIZACION DE LA PANTALLA Y DEL TECLADO 177
La lectura del teclado por el ordenador. La forma efectiva de utilizar el salto relativo condicional. Cómo se provoca un mensaje de error. La generación de caracteres y de gráficos de alta resolución. La división de la pantalla en tres secciones diferentes. La utilización de las rutinas en CM en un programa en BASIC. Una rutina para crear los Gráficos Definidos por el Usuario.

CAPITULO 10: LA PILA DE MEMORIA (STACK) 195
El «puntero de pila». La utilización de una pila dentro de un programa. El juego de «Las Torres de Hanoi». El acceso a una pila del BASIC a través del Código Máquina. El traslado del contenido de un bloque de memoria a otro bloque. La utilización del bucle en los traslados. Simplificación de rutinas utilizando los comandos «INC», «DEC», «LDIR» y «LDDR». Una rutina para desplazar todo el contenido de la pantalla. Un comando de velocidad increíble.

APENDICE A 207
Tablas de conversión HEX/DECIMAL/BINARIO

APENDICE B 215
Códigos de operación del Z80 clasificados por mnemónicos.

APENDICE C 231
Instrucciones/Tabla de señalizadores.

EL AUTOR

James Walsh, en el momento de escribir este libro, tiene dieciséis años y estudia en el Davenant Foundation School en Loughton, Essex.

Actualmente ha alcanzado gran prestigio por sus colaboraciones en varias revistas de ordenadores de ámbito nacional en Gran Bretaña, así como sus previas aportaciones en la publicación de dos libros sobre ordenadores Sinclair.

Además de diversos artículos, de sus análisis de software y de la publicación de libros, todavía le sobra tiempo para ser un buen aficionado a la fotografía, ser un músico mediocre y, sobre todo, constituir una preocupación para sus profesores.

Espera publicar su próximo libro en un futuro no muy lejano —si los exámenes se lo permiten— aunque su ambición más inmediata es la de viajar al extranjero.

El tiempo dedicado a escribir este libro ha facilitado muy poco su vida social. Ahora espera encontrar tiempo para corregir esta deficiencia. Para finalizar, reconoce la paciencia y el apoyo de toda su familia y amigos.

Clive C. Walsh

INTRODUCCION

El objeto de este libro es el de ofrecer una introducción coherente y fácilmente asimilable de la programación del Código Máquina, considerada necesaria partiendo de la base de que el autor hubiera deseado tenerla a su alcance cuando comenzó a pelear con este tema cargado de misterio. Hasta más tarde no se dio cuenta de que tal misterio, en realidad, no existía.

Se han realizado todo tipo de esfuerzos para evitar caer en la típica trampa, tanto en lo que se refiere a los conceptos que se dan «por entendidos» por parte del lector como, por otro lado, presumir que es un ignorante total. De todas formas ha sido necesario realizar una aproximación al tema desde el punto de vista del «principiante», tratando siempre de anticiparnos a los problemas que él o ella puedan hallar.

Si encuentras el libro demasiado fácil, entonces probablemente no se haya escrito para ti. Si lo encuentras demasiado difícil entonces te pido perdón ya que debo presumir que a estas horas estarás pidiendo ayuda.

Los ordenadores tienen su propio idioma

Capítulo 1

¿POR QUE EL CODIGO MAQUINA?

La respuesta a esta pregunta es bastante sencilla: el Código Máquina es el lenguaje propio del ordenador. Por la misma razón, «tiene más sentido para el ordenador». Podría ofrecerte una definición más compleja y técnicamente más correcta del Código Máquina pero probablemente te dejaría peor informado que cuando empezaste. De todas formas, este libro representa un intento de hacer desaparecer el misterio y la confusión que rodea al asunto de la comunicación con los ordenadores y, en particular, con tu Spectrum.

Por lo tanto, todos los esfuerzos se han realizado con el objeto de dar la información y las explicaciones de la forma más sencilla posible. Si, en el caso de que hayas realizado la lectura total del libro, llegas a tener un gran dominio de las ideas esenciales del Código Máquina y son lo suficientemente fiables para utilizarlo, entonces habrá valido la pena el esfuerzo.

La mejor tarea que realizan los ordenadores es la de sumar «uno» y «uno» y finalmente llegar a «dos», que es la respuesta correcta. Esto no es realmente muy inteligente ni muy impresionante. Sin embargo, parecen tener una capacidad infinita para intimidar a los seres humanos y esto puede convencernos fácilmente de que los ordenadores son demasiado complicados, demasiado rápidos, demasiado inteligentes o, simplemente, dan demasiado miedo como para enfrentarse con ellos.

A pesar de no parecer lo contrario, tu Spectrum no es otra cosa que una combinación de componentes electrónicos los cuales, colectivamente, tienen capacidad para procesar la información que le suministremos de acuerdo con las instrucciones que le ordenemos. Aunque el resultado final es un conjunto de piezas perfeccionadas, se trata de una herramienta para ser utilizada pero es una herramienta que, al igual que todas, no tiene conciencia ni voluntad por sí misma.

Solamente puede trabajar con lo que le suministramos y solamente puede hacer lo que hace porque «tú» le dices lo que tiene que hacer. De todos modos puede llevar a cabo sus objetivos y tareas con más eficacia, con mayor seguridad y con menos tiempo que el cerebro humano.

¿Por qué afrontar el problema de entender la naturaleza del Código Máquina y luego perder el tiempo tratando de aprender a utilizarlo? ¿Es que no tienes ya un «lenguaje» —el BASIC— perfectamente adecuado con el que comunicarte con tu Spectrum? Ciertamente el BASIC es un lenguaje con el que, a estas alturas, ya estás familiarizado y es un lenguaje que, al menos, parece razonablemente inteligible (por lo menos para los angloparlantes). De hecho, eso es precisamente lo que es: un lenguaje de ordenador escrito para los humanos. Su objeto es permitirnos a todos «hablar» con nuestro ZX80/ZX81/Spectrum de una manera compacta y ordenada, utilizando el lenguaje de tal modo que lo comprendamos y que, también, pueda ser «interpretado» por el ordenador. Si, «interpretado». Dentro de la memoria del ordenador hay un diccionario completo de BASIC al Código Máquina. Por lo tanto, el ordenador acude a este diccionario para traducir y para entender cualquier cosa que se le introduce o suministra en BASIC, antes de que pueda llevar a cabo las instrucciones correspondientes.

Por consiguiente, si el objetivo es alcanzar un mejor entendimiento y tener una respuesta más rápida de tu Spectrum, si nos es posible «hablar en su mismo idioma», entonces se trata de algo muy importante. Es fácil imaginarse lo difícil que te sería comunicarte si, cuando viajas a un país extranjero, te encontraras con alguien que solamente hablara su propio idioma «extranjero» y tú no tuvieras ni idea de cómo se habla tal idioma. Como no recurriéramos a un lenguaje de signos que fuera inteligible, nos sería prácticamente imposible progresar algo y, de hecho, el lenguaje de signos

o símbolos sería un esfuerzo colectivo para encontrar un lenguaje común y sencillo que pueda ser entendido por todos.

La función del intérprete en el ordenador

Si podemos beneficiarnos de los servicios de un intérprete, entonces el proceso de comunicación empezará a resultar un poco más fácil. De todas formas cada cosa que pasa entre tú y el ordenador tendrá que ser traducida y luego interpretada de forma que la «conversación» inevitablemente puede ser bastante limitada y muy lenta.

Esta es la situación en la que se encuentra el Spectrum: tiene un intérprete incorporado. Cualquier cosa que se introduce en BASIC es traducida a su propio lenguaje y viceversa siendo, por tanto, un límite natural respecto a tu comunicación con él y, también, un límite en cuanto a la velocidad de dicha comunicación. Cuando se escriben programas en BASIC, antes de operar con ellos, el Spectrum primero tiene que almacenarlos y luego interpretar todos los comandos y los datos. La «interpretación» o traducción al Código Máquina tiene por objeto permitir el funcionamiento del microprocesador Z80 ya que es el UNICO lenguaje que es capaz de entender. Inevitablemente, este proceso de *búsqueda,* gracias a la posibilidad de almacenamiento del ordenador, *localizando* las instrucciones y los datos para *ejecutarlos* posteriormente, lleva un tiempo. Si a ello le añadimos que cada cosa debe ser también traducida del BASIC al Código Máquina antes de empezar a trabajar, entonces el tiempo que emplea es mucho mayor. Hay veces en las que este hecho puede ser particularmente importante, por ejemplo, donde el control de los gráficos o la velocidad de reacción del programa son fundamentales.

Del mismo modo que cuando viajas al extranjero, finalmente deseas ser capaz de comunicarte con fluidez, facilidad y de forma acertada, para lo cual no existe cosa mejor que aprender el lenguaje o idioma del país por el que viajas. Por ahora, has elegido viajar dentro del territorio del propio ordenador y, por lo tanto, debes aprender su idioma. Puede parecer descabellado pero créeme: el Spectrum no va a aprender el tuyo ¡ni ahora ni nunca!

Mayor velocidad en Código Máquina que en BASIC

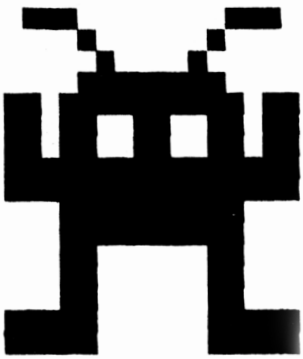
Por supuesto, aprender un nuevo «idioma» extranjero no resulta tan fácil. Afortunadamente el idioma del ordenador es extremadamente sencillo, aunque la forma en la que el ordenador lo utiliza ya es otra historia, pero esto lo veremos conjuntamente. El ordenador es muy hábil sumando «uno y uno» y esto es realmente todo lo que puede hacer. Se pasa toda su existencia tratando con «ceros» y «unos».

Para entender por qué el ordenador sólo trata con «ceros» y con «unos» y cómo este sencillo fundamento le permite realizar las tareas más complicadas, es necesario tener algún conocimiento acerca de la manera en que está montado el ordenador y cómo hace luego para trabajar. Antes de ver esto, vamos a echar una ojeada a la velocidad con que puede responder a las instrucciones dadas en Código Máquina en lugar de las suministradas en BASIC.

Escribe este programa:

```
   5 BRIGHT 1: PAPER 4: BORDER 5
: INK 1: CLS
  10 PRINT AT 5,8;"
"
  15 PRINT AT 10,0;"
"
  20 PRINT AT 13,0;"
"
  25 PRINT AT 0,0;"ALMACENANDO E
L DIBUJO "
  30 FOR X=0 TO 7167
  40 POKE (25600+X),PEEK (16384+
X)
  50 NEXT X
  60 PRINT AT 0,0;"FIN DEL ALMAC
ENAMIENTO"
  70 PAUSE 100
  80 CLS
  90 PRINT "RECUPERACION DEL DIB
UJO"
 100 FOR X=0 TO 7167
 110 POKE (16384+X),PEEK (25600+
X)
 120 NEXT X
 130 PRINT "FIN DEL PROGRAMA"
```

ALMACENANDO EL DIBUJO



¿A qué resulta bastante lento? Ahora escribe el programa en la versión equivalente en Código Máquina tal y como se indica más abajo y fíjate en la diferencia.

Primero borra las líneas 30, 40, 50, 70, 100, 110 y 120 y luego añade las siguientes:

```
 150 STOP
 200 FOR A=25501 TO 25524: READ
A$
 210 LET TOT=(CODE A$(1)-48)*16
 220 IF TOT>9*16 THEN  LET TOT=T
OT-7*16
 230 LET B=CODE A$(2)-48: IF B>9
 THEN  LET B=B-7
 240 LET TOT=TOT+B: POKE A,TOT
 250 NEXT A
 300 DATA "21","00","40","11","0
0","64","01","00","1C","ED","B0"
,"C9"
 310 DATA "21","00","64","11","0
0","40","01","00","1C","ED","B0"
,"C9"
```

... y ejecuta.

```
RUN 200
```

Ahora añade esta línea:

```
1000 CLS : PAUSE 30: RANDOMIZE U
SR 25513: PAUSE 30: GO TO 1000
```

... y ejecuta.

```
RUN 1000
```

¿Te has convencido ya?

El ordenador visto como un interruptor eléctrico

Volvamos a la forma en la que actúa el ordenador. Lo que éste realiza esencialmente es hallar los cálculos aritméticamente, utilizando un sistema de aritmética binaria y una memoria en la cual se almacena la información. Tu Spectrum, como ordenador digital que es, sólo reconoce DOS estados o posiciones; es capaz de reconocer, bien la presencia, bien la ausencia de un impulso eléctrico. Por lo tanto, opera sobre la base de encontrarse en la posición «on» o en la posición «off»*, y de ahí la expresión «sistema binario» y la naturaleza biestable del ordenador.

Una vez conocida esta información tan importante podremos dar el paso siguiente, que es el de representar estos dos estados mediante la utilización de «ceros» y «unos». Se trata de una progresión lógica que representa la «ausencia de señal eléctrica» como un «0», y la «existencia de señal eléctrica» como un «1». Los dos estados estables del ordenador, por lo tanto, se pueden representar como:

1. ¿Hay alguna señal? Sí — represéntalo como un «1».
2. ¿Hay alguna señal? No — represéntalo como un «0».

---

* *N. del T.:* Debido a la gran difusión de estos dos términos, tanto en el uso de aparatos domésticos como en el mundo de la informática, se ha preferido, para una mayor claridad de la obra, mantenerlos a lo largo de la misma. «On» y «off» significan «conectado» o «encendido» y «desconectado» o «apagado», respectivamente.

La comparación más sencilla de cómo funciona esto es la relativa a la función que desempeña un interruptor eléctrico, el cual puede estar en la posición de conectado («ON») o en la de desconectado (Apagado) («OFF»). Cuando el interruptor está en la posición «ON», entonces la corriente eléctrica puede circular, cuando está en la posición «OFF», por el contrario, ésta no puede circular. Puede parecer que hemos cogido al ordenador, que es algo que, por lo menos, puede sumar hasta dos, y que lo hemos reducido a algo similar a un vulgar interruptor casero de la luz. Hasta cierto punto esto puede ser efectivamente cierto pero, por descontado, esto haría disminuir de forma espectacular las ventas de los pobrecitos y viejos ordenadores. De hecho, tu Spectrum o, más bien, su microprocesador Z80, tiene capacidad suficiente como para formar series de cadenas de las mencionadas decisiones ON/OFF y de producir un número bastante elevado de combinaciones diferentes, cada una de las cuales tiene su propio significado.

Las «combinaciones» de decisiones, una muestra de las posibilidades del ordenador

Para entender esto de una manera más completa tomemos, por ejemplo, dos de estos «cambios» sencillos ON/OFF y vamos a combinarlos. Esto nos permitirá ver cuatro estados diferentes:

1. ON y ON ..... se convierten en 11
2. ON y OFF..... se convierten en 10
3. OFF y ON..... se convierten en 01
4. OFF y OFF..... se convierten en 00

La combinación de tres de estos «cambios», los cuales siguen siendo simples representaciones de los estados ON/OFF reconocidos por el ordenador, nos permite obtener ocho estados específicos diferentes:

1. OFF y OFF y OFF 000
2. OFF y OFF y ON 001
3. OFF y ON y ON 011

4. OFF y ON y OFF 010
5. ON y ON y ON..... 111
6. ON y ON y OFF..... 110
7. ON y OFF y OFF 100
8. ON y OFF y ON 101

Lo que empieza a resultar significativo es la regla por la que se rigen estas «combinaciones». Si tomamos dos estados concretos, éstos pueden combinarse en un máximo de 2 × 2 (es decir, cuatro) formas diferentes. Si tomamos tres y los combinamos entre sí, el máximo será de 2 × 2 × 2 (ocho) posibles combinaciones diferentes.

Ahora y de la misma forma que somos capaces de encadenar y unir las letras del alfabeto, y hacer combinaciones específicas a las que llamamos «palabras», cada una con su distinto significado, el ordenador también puede encadenar combinaciones de los dígitos «0» y «1», lo cual da lugar a «palabras» identificables y diferenciadas unas de otras. Tu Spectrum puede manejar «palabras» compuestas por ocho dígitos y, utilizando la regla de las combinaciones, se puede observar que se nos ofrece un «vocabulario» de 256 «palabras» diferentes (2 × 2 × 2 × 2 × 2 × 2 × 2 × 2 ó «dos elevado a la octava potencia»). Al encadenar o unir estos dos dígitos de esta forma, las «palabras» se construyen de manera que el ordenador pueda entenderlas. Cada «palabra» se conoce por el nombre de «número binario». El nombre asignado a ese número binario es el de BYTE (se lee «bait»), y los dígitos que lo componen son los llamados BITS (Fig. 1).

En este capítulo de introducción se ha intentado que tú, lector, empieces a ir conociendo los términos manejados por tu ordenador, tanto los sencillos como los complicados. El Código Máquina es

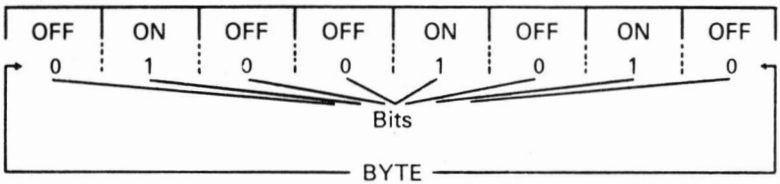


Figura 1

el lenguaje que utiliza y, por lo tanto, entiende y, como has podido observar, este lenguaje se puede reducir a cadenas de ceros y unos.

El próximo capítulo tratará de una forma elemental o, por lo menos, resumida, de cómo están organizados los distintos componentes del ordenador, con el fin de ir comprendiendo algo de lo que hace cuando trabaja, y cómo es utilizada la «aritmética binaria» para realizar esta función. El próximo objetivo será ver cómo son «empaquetadas» las instrucciones antes de ser ofrecidas al ordenador en forma de palabras que pueda entender y que, a su vez, podamos manejar.

Más adelante será necesario tratar el asunto de cómo «interpretamos» un número binario por nuestros propios medios, para lo cual no será necesario resignarse a la horrible perspectiva de tener que leer series interminables de ceros y unos, o tener que alimentar al ordenador con una dieta parecida.

Si los modales caracterizan a los hombres, los números definen al ordenador...

Capítulo 2
PALABRAS FORMADAS POR NUMEROS

El dibujo de la figura 2 es una sencilla representación esquemática en la que se muestra la organización interna del Spectrum. Ya hemos hecho referencia al microprocesador Z80 y a sus funciones de BUSQUEDA, LOCALIZACION y de EJECUCIÓN. Este microprocesador, que es el que contiene la Unidad Central de Proceso (CPU), como su propio nombre indica, es el componente central del ordenador. Tiene comunicación bidireccional con los otros dos dispositivos: el ENLACE de entradas/salidas (INPUT/OUTPUT) y la MEMORIA.

El enlace del ordenador con dispositivos exteriores

El ENLACE: permite la «salida» de los resultados obtenidos por la CPU en su trabajo y enviarlos hacia otros dispositivos, tales como

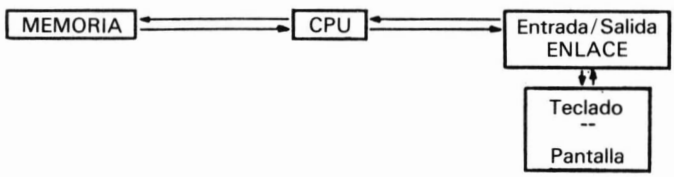


Figura 2

la pantalla de televisión o un monitor de vídeo, con el objeto de hacer una presentación visual de esos resultados. De modo alternativo puede lograr que esos resultados permanezcan almacenados en la cinta cassette de un grabador electromagnético o en otro tipo de almacenamiento más refinado como es el de guardarlos en un disco magnético.

El ENLACE también es capaz de conseguir que la información sea suministrada («entrada») desde fuera del ordenador, por ejemplo, la que proviene del teclado o la introducida desde un fichero externo. También puede utilizar este mecanismo para realizar, por ejemplo, conexiones con circuitos que dispongan de cierto tipo de sensores y, por consiguiente, puede hacer todo lo necesario para controlar la producción de una fábrica o interrumpirte el juego de marcianitos para avisarte de que ¡se te ha quemado la comida! De aquí que el Enlace sea conocido como la línea de ENTRADA/SALIDA (INPUT/OUTPUT) del ordenador.

Dos tipos de memoria: la RAM y la ROM

El otro componente consiste en la llamada MEMORIA. Realiza exactamente lo que te imaginas: almacenar datos e información los cuales pueden ser recuperados más adelante. Debido a su fiabilidad recuerda todo aquello que se le introduce y es a esta Memoria donde se dirige la CPU cuando necesita encontrar lo que se le pide. Esencialmente hay dos tipos de Memoria: la RAM y la ROM.

RAM es la abreviatura perteneciente a **R**andom **A**ccess **M**emory (Memoria de Acceso Aleatorio), lo que significa que si deseas leer o escribir, al principio o al final de la memoria, le lleva exactamente la misma cantidad de tiempo. Esto sucede en contraste directo con otros dispositivos de almacenamiento, tales como una cinta cassette o un disco magnético en los cuales, para encontrar la parte final de una información, primero es necesario leer toda la información precedente. De manera análoga, con una cinta grabada, es posible cargar la RAM (introducir en la memoria) así como extraer información (leer la memoria) de la misma.

Por el contrario, ROM significa **R**ead **O**nly **M**emory (Memoria de Sólo Lectura). Como ya te puedes imaginar, esto indica que sólo es posible leer cuál es el contenido de esta parte de la Me-

moria, sin la posibilidad de poder cambiarla a nuestra voluntad. Es parecido al sistema con el que nos es posible leer (escuchar) la información (la música) que está almacenada en un L.P. musical, en el que nos resulta imposible añadir ninguna nueva y reciente información (¡a no ser que lo hagamos por medio de un acto de auténtico vandalismo!).

Desde luego, puedes intentar cambiar esa información, pero sería lo mismo que apretar el botón de grabación de un magnetofón sin haber colocado antes la cinta en su interior. Funcionarán los engranajes pero no pasará nada. Para una mayor claridad, es necesario decir que también la ROM es una memoria de acceso aleatorio en el mismo sentido y en la misma forma que la RAM (en lo que se refiere al tiempo de acceso a la misma), la única diferencia es que la ROM se define de forma más acertada como una **R**andom **A**ccess **R**ead **O**nly **M**emory (Memoria de Sólo Lectura de Acceso Aleatorio).

Hasta ahora he hecho todo lo posible para evitar el tradicional «planteamiento numérico» de los ordenadores y de la Informática, con el fin y con la esperanza de mantener vivo tu interés. A pesar de todo, los números tienen su lugar legítimo en el esquema de las cosas y, por consiguiente, no pueden ser ignorados. El enfoque que vamos a abordar trata acerca de cómo los números son almacenados y utilizados, lo que nos proporcionará la base esencial para entender la manera de utilizarlo y de comunicarnos con el Spectrum como si se tratara de un todo integrado, en lugar de ser un surtido de componentes separados.

Del sistema decimal al binario

Volviendo a referirnos a la analogía de los cambios ON/OFF, veamos la figura 3.

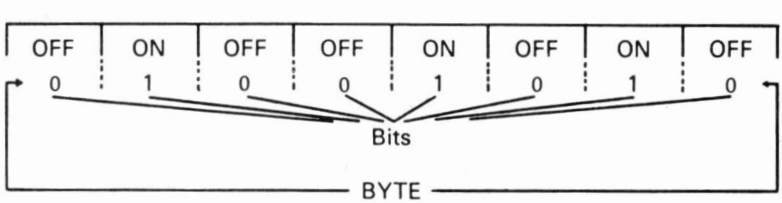


Figura 3

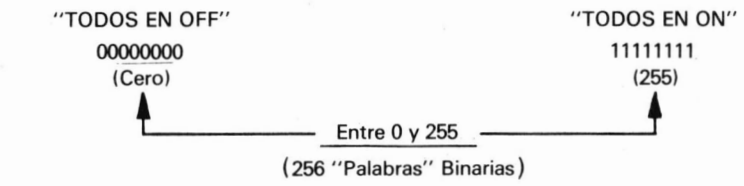


Figura 4

Se puede observar que el número binario 01001010 es una «palabra binaria» específica que cae dentro del campo del vocabulario de 256 palabras del Spectrum. Los límites de ese campo están representados por «todas las posiciones en OFF» (apagadas) y «todas las posiciones en ON» (encendidas o conectadas), como se ve en la figura 4.

Entonces, ¿por qué una cadena de ocho nadas son el «cero» y ocho unos son equivalentes al número decimal 255? La respuesta reside en el tipo de aritmética utilizada por el ordenador: ARITMETICA BINARIA, la cual quizás sea mejor entendida si primero echamos una ojeada a la estructura del sistema «decimal» al cual estamos acostumbrados.

El Sistema Decimal está basado en el empleo de diez dígitos (del 0 al 9) por lo que también es designado como «BASE DIEZ». También utiliza un sistema de «Notación Posicional». Todo ello permite calcular el valor de cualquier dígito dentro de un número, sobre la base de su posición en relación al Punto Decimal de dicho número. Como cuestión de precisión, la notación posicional o la notación de punto descansa en la posición de un dígito en relación al punto unidad.

En aritmética binaria, por ejemplo, hablaremos del *punto binario.* El *punto unidad* es el punto (o la coma) que separa la *parte entera* de la *parte fraccionaria* de un número. Continuando con el sistema decimal, el valor de un dígito (o cifra) dentro de un número es igual al producto de este dígito por la «potencia de 10», determinada por su posición en relación al punto decimal. Los dígitos de un número están dispuestos en «columnas», y cada columna tiene un valor diez veces mayor que la columna situada inmediatamente a su derecha (Gráfico A).

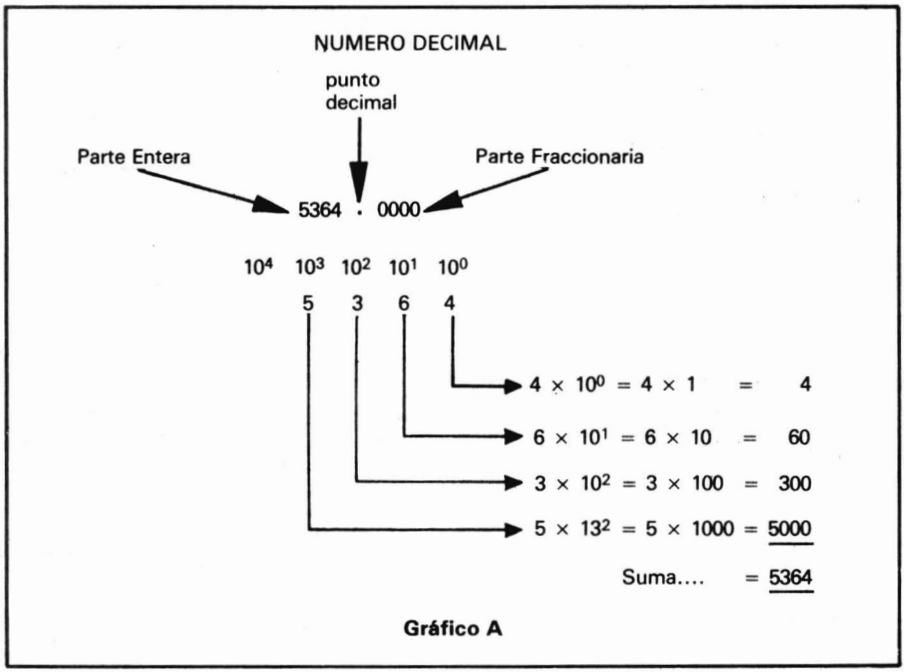


Gráfico A

Cada dígito/columna tiene una «potencia» diez veces mayor que la del dígito/columna situada inmediatamente a su derecha (ver Gráfico B).

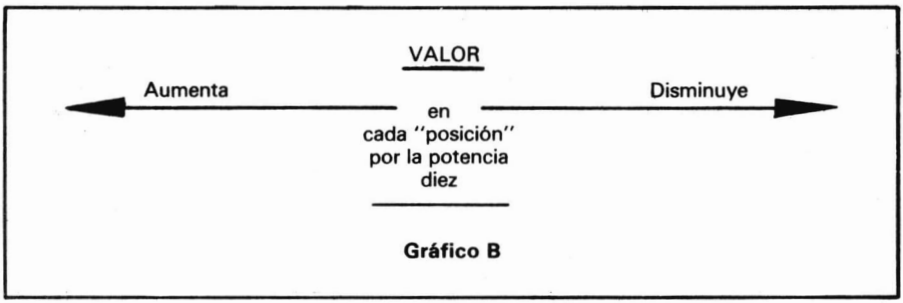


Gráfico B

El sistema binario está basado en sólo dos dígitos (0 y 1) y es, por lo tanto, de BASE DOS. Al igual que en el sistema decimal, se utiliza un método de notación posicional, aunque cada dígito dentro de un número binario ahora tiene un valor que es una «po-

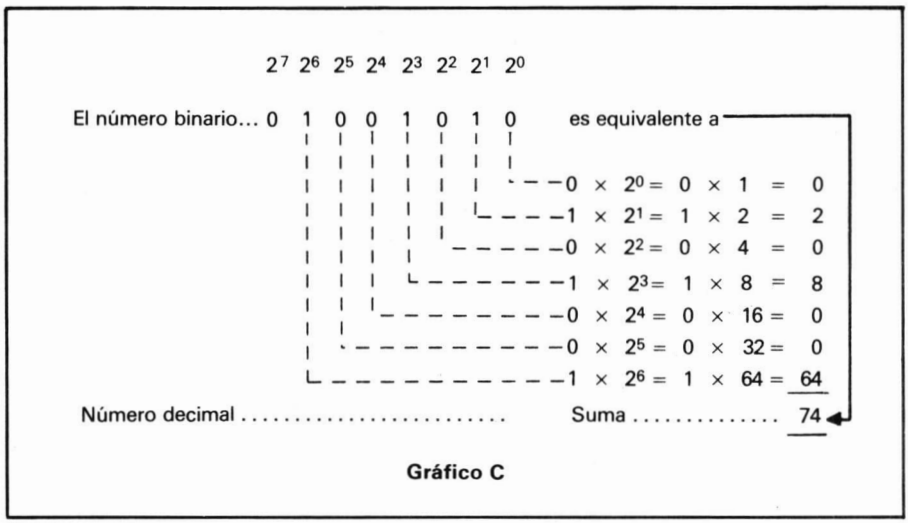


Gráfico C

tencia de dos» mayor que el situado inmediatamente a su derecha (Gráfico C).

Se puede observar que en el ejemplo arriba indicado 01001010 es el binario equivalente al decimal 74. Del mismo modo 00000000 es igual a cero (decimal), y 11111111 = 255 (decimal), como se ve en el Gráfico D.

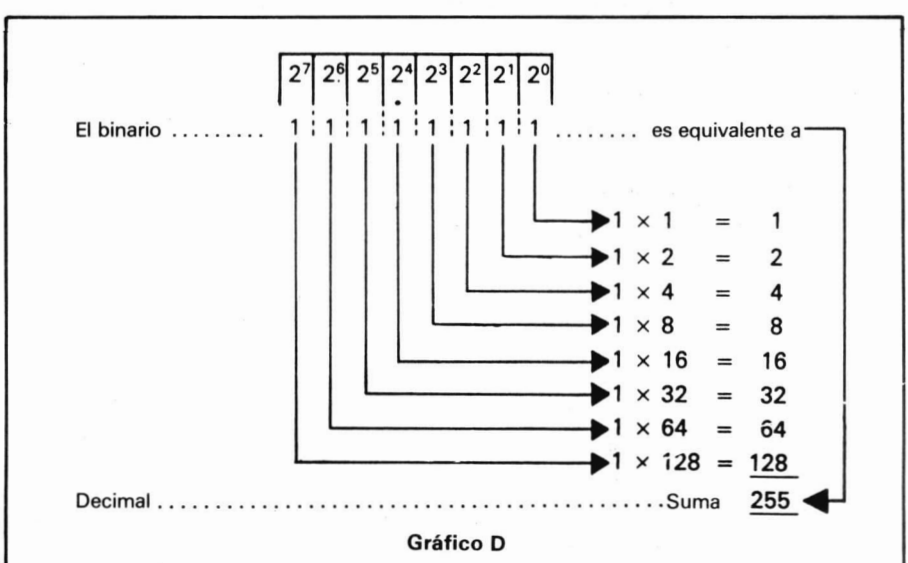


Gráfico D

El sistema hexadecimal y su utilidad

Mientras que el ordenador utiliza el número binario equivalente al número decimal con el que se va a operar, yo te había prometido en el capítulo precedente que no te sería necesario recordar y usar cadenas de ceros («0») y unos («1») para comunicarte con el Spectrum. Desgraciadamente tampoco podrás acudir a los tan familiares números decimales. De todas formas, si resulta que la «base 10» se va a utilizar poco y la «base 2» no es práctica, resulta que la «base 16» va a ser la única aceptable. ¡No desesperes!, la verdad es que resulta más fácil de lo que parece.

Si aplicamos los principios recién vistos, la base 16 (o HEXADECIMAL) emplea un sistema de dieciséis dígitos. Los diez primeros de la secuencia hexadecimal se corresponden con los del sistema decimal (del 0 al 9), los seis restantes están representados por las seis primeras letras del alfabeto (A, B, C, D, E, F). La tabla 1 nos muestra la relación Binario/Decimal/Hexadecimal.

TABLA 1

Número Binario	*Equivalente Decimal*	*Equivalente Hexadecimal*
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

En el Apéndice A puedes encontrar una tabla completa con las equivalencias Binario/Decimal/Hexadecimal desde 0 hasta 255.

Tomemos de nuevo el anterior ejemplo del número 74 (decimal) y su equivalente 01001010 (binario) y fíjate a través del Gráfico E en la manera en que se va a convertir en un número hexadecimal (frecuentemente acortado a la expresión «Hex» por comodidad):

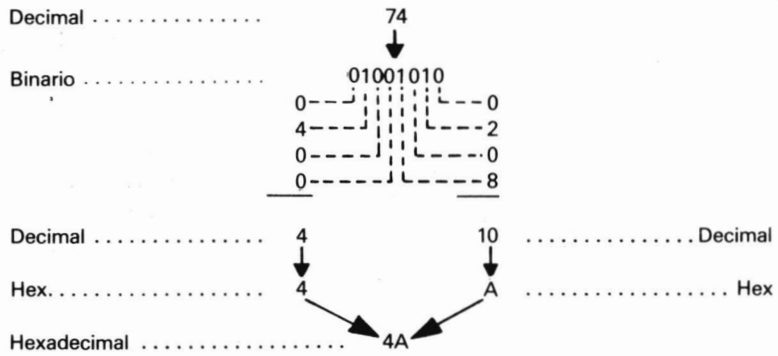
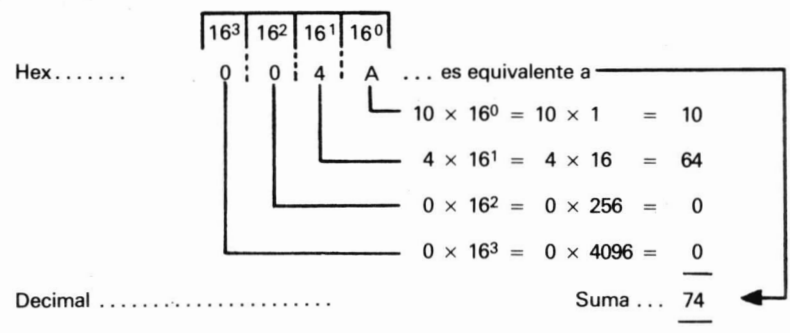


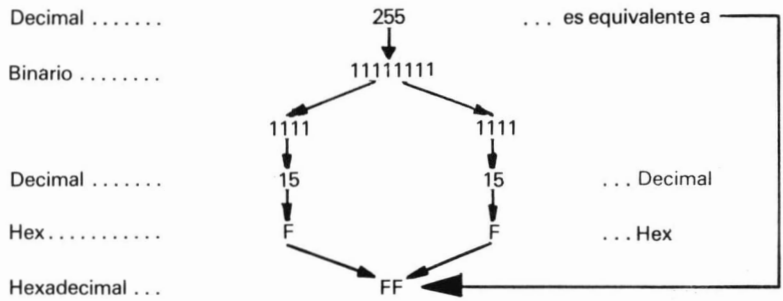
Gráfico E

Comprueba (Gráfico F):



Número binario	*Equivalente decimal*	*Equivalente Hexadecimal*
01001010	74	4A

Gráfico F



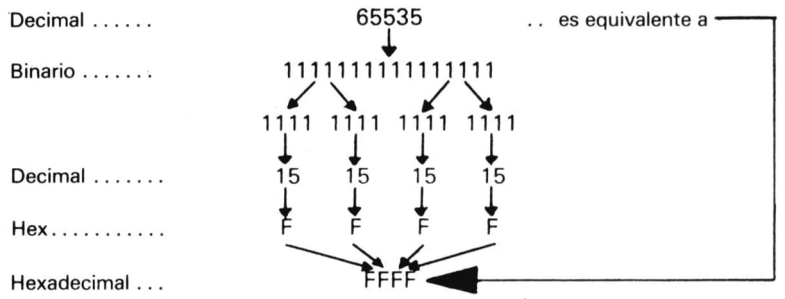
Número binario	*Equivalente decimal*	*Equivalente hexadecimal*
11111111	255	FF

Gráfico G

Si hacemos lo mismo con 255, que es el número mayor que puede manejar la CPU, veremos con qué facilidad cualquier número binario de 8 bits puede ser representado por un número Hex de dos dígitos, como muestra el Gráfico G.

Este «agrupamiento» de los dígitos binarios en series de cuatro junto con la combinación de sus Hex equivalentes, resulta ser el método más conveniente e importante para la representación de las DIRECCIONES o POSICIONES DE MEMORIA. El hecho de que el 255 sea el número más alto que puede ser manejado ha llegado a crear grandes problemas. Por otro lado, esta restricción no es absoluta y en breve te demostraré cómo la CPU del Spectrum es capaz de utilizar su habilidad para sumar 1 y 1, y llegar hasta ¡¡¡65536!!!

El manejo de un campo de números desde el 0 hasta el 65535 precisa de un número binario de 16 bits, el mayor de los cuales será el 1111111111111111 que es el binario equivalente de 65535 (compruébalo si tienes alguna duda). Todos los números afectados, ya sean binarios o decimales, se han vuelto embarazosamente largos. El valor concreto de un número hexadecimal se volverá un poco más complicado. Bueno, no tanto ya que nos permite disponer a cada número binario en grupos de cuatro dígitos. Cada grupo, a su vez, puede representarse por un hexadecimal de un dígito y éste, en



Número binario	Equivalente decimal	Equivalente hexadecimal
1111111111111111	65535	FFFF

Gráfico H

combinación con los otros tres dígitos Hex correspondientes a los otros grupos, formarán un número Hex de cuatro dígitos (o cifras) equivalente a cualquier número binario de 16 bits (Gráfico H).

Todos los números comprendidos entre 0 y 65535 pueden ser reducidos a un número Hex de cuatro dígitos entre 0000 y FFFF.

Los cálculos en aritmética binaria

Como en todos los «sistemas de numeración», los cálculos aritméticos pueden realizarse en aritmética binaria y, por supuesto, el ordenador es muy partidario de esta costumbre. Todas las operaciones se realizan sobre la base de la suma o de la resta —el Spectrum todavía funciona de tal forma que realiza las substracciones por medio de «sumas», pero esto lo veremos dentro de poco. Al menos por ahora, contamos con la ventaja de no tener que preocuparnos acerca de las funciones de multiplicación y división. En términos generales, los convenios (o reglas) que se aplican a la aritmética decimal sirven también para la aritmética binaria. Así, el número más alto que puede tener lugar en una columna es 1; esto

es, uno menos que el número total de dígitos permitidos para ese sistema (dos menos uno). En el decimal el dígito mayor es 10 — 1 = 9. Cuando realizamos sumas entre dos números binarios la regla familiar de «llevar una a la columna siguiente» (acarreo) se utiliza cuando la suma de los dos dígitos es mayor que 1. Hacemos una referencia al Gráfico I que clarificará estas «verdades» básicas:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10 ("cero y llevo una")

+	0	1
0	0	1
1	1	10

Gráfico I

Un ejemplo de suma de dos números binarios presentará el aspecto del Gráfico J:

	01001010	(Decimal 74)
	+ 00111010	(Decimal 58) +
Acarreo	0111010—	=
Suma	10000100	(Decimal 132)

Gráfico J

Aplicando los mismos principios a la resta, ésta se realiza mediante el proceso inverso al de la suma (tal y como se haría en una resta normal, sólo que del 1 al 0 necesitamos una, y llevamos una) (véase Gráfico K).

	10000100	(Decimal 132)
	—00111010	(Decimal 58) —
Acarreo	01111010—	=
	01001010	(Decimal 74)

Gráfico K

En realidad se trata de un proceso demasiado complicado, por lo que las restas son más fáciles de resolver utilizando un método conocido como «El complemento a dos» (o «número complementario»), lo cual es lo, que en realidad, hace el ordenador. En el ejemplo anterior el resultado de la resta puede conseguirse más fácilmente AÑADIENDO «el complemento a dos» de 00111010 (el sustraendo) a 10000100 (el minuendo). Para determinar el «complemento a dos» de cualquier número binario simplemente basta cambiar todos los 0 por 1, y todos los 1 por 0, y luego sumarle 1 al resultado, como se muestra en el Gráfico L.

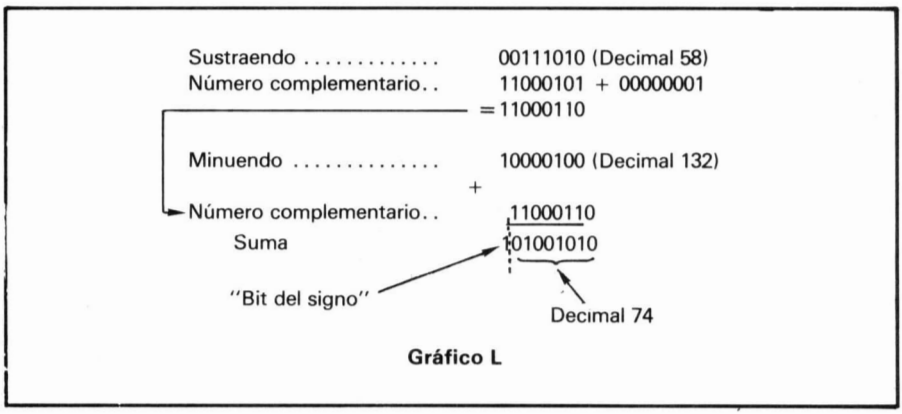


Gráfico L

Los ocho últimos bits son los mismos de antes (74 decimal), aunque ahora hay un «bit de acarreo» extra, conocido también como «bit del signo». Este último bit puede ser omitido a la hora de determinar el resultado numérico pero se utiliza como indicador según que el resultado del cálculo sea positivo o negativo. Cuando el bit de acarreo vale 1 significa que el número resultante tiene valor positivo. A la inversa, cuando el bit de acarreo es 0, el resultado de la operación tiene valor negativo.

Los ordenadores trabajan formando edificios

Capítulo 3

EL DIRECCIONAMIENTO DE LAS POSICIONES EN LA MEMORIA DEL SPECTRUM

Será interesante repasar todos los puntos que hemos visto hasta ahora. Empezamos con una descripción muy general sobre el Código Máquina como lenguaje propio del ordenador y hemos considerado las ventajas que podemos conseguir con el aprendizaje de dicho lenguaje. De igual modo que nos hacíamos una idea general acerca de cómo funciona el Spectrum, comenzamos a observar con más detalle cómo se relacionan entre sí sus distintos componentes y utilizan la aritmética binaria para trabajar. Esto ha hecho necesario que nos enfrentemos con otro sistema numérico (el hexadecimal) con el propósito de aprender a reducir números binarios de 8 bits a uno Hex de 2 bits, o números binarios de 16 bits a números Hex de 4 bits.

Ya hemos visto el tamaño del vocabulario limitado del Spectrum que contiene 256 «palabras». También he comentado su capacidad para almacenar información en más de 65000 posiciones distintas. Este sistema de «Direccionamiento» será el que estudiaremos a fondo a lo largo de este capítulo. Además examinaremos con mucho más detalle la forma en que está estructurada y organizada la me-

moria del Spectrum, y empezaremos el aprendizaje necesario para manipular el contenido de esa memoria.

Estructura y organización de la memoria del Spectrum

Nuestro primer objetivo consiste en explorar la estructura y la organización de la memoria del Spectrum haciendo referencia al mapa de memoria (Fig. 5).

Cada posición de la memoria es un lugar en el que puede almacenarse la información y es comúnmente conocida por el nombre de DIRECCION. Esta terminología no ha sido creada por azar sino que se trata de la utilización de un concepto que ya nos es familiar. Cada posición tiene un número asignado entre el 0 y el 65535. De esta manera, el número nos da la dirección específica de la posición. Una ojeada al mapa de memoria nos revela que las primeras 16384 (ó 16K) direcciones están ocupadas por la ROM. Esta, como ya sabemos, es la Memoria de Sólo Lectura y, esencialmente, es un bloque de 16384 posiciones en las que la información es almacenada en el momento de la fabricación de la pastilla (Chip) de ROM. No es, por tanto, susceptible de ser cambiado o alterado por el usuario. En la versión del Spectrum de 16K hay, además, 16384 posiciones de RAM. La versión de 48K de este microordenador utiliza el máximo número de posiciones que pueden ser direccionadas por la CPU del Z80. Por consiguiente, además de las 16K de ROM, hay 48K adicionales de RAM (posiciones que van desde la 32767 a la 65535 o, si se expresa en Hex, desde 8000 a FFFF). La combinación de las 16K de ROM y las 48K de RAM nos da un total de 64K de memoria ó 65536 bytes.

La memoria considerada como un edificio de viviendas

Antes de continuar, prefiero disipar cualquier duda o confusión que pueda existir en relación al número «65535». A estas alturas debe quedar claro que la CPU del Spectrum puede manejar 256 bytes diferentes de 8 bits. Ahora vamos a considerar, hipotéticamente, que la memoria es como un edificio de apartamentos de 256 pisos de altura, y que en cada planta hay 256 viviendas independientes.

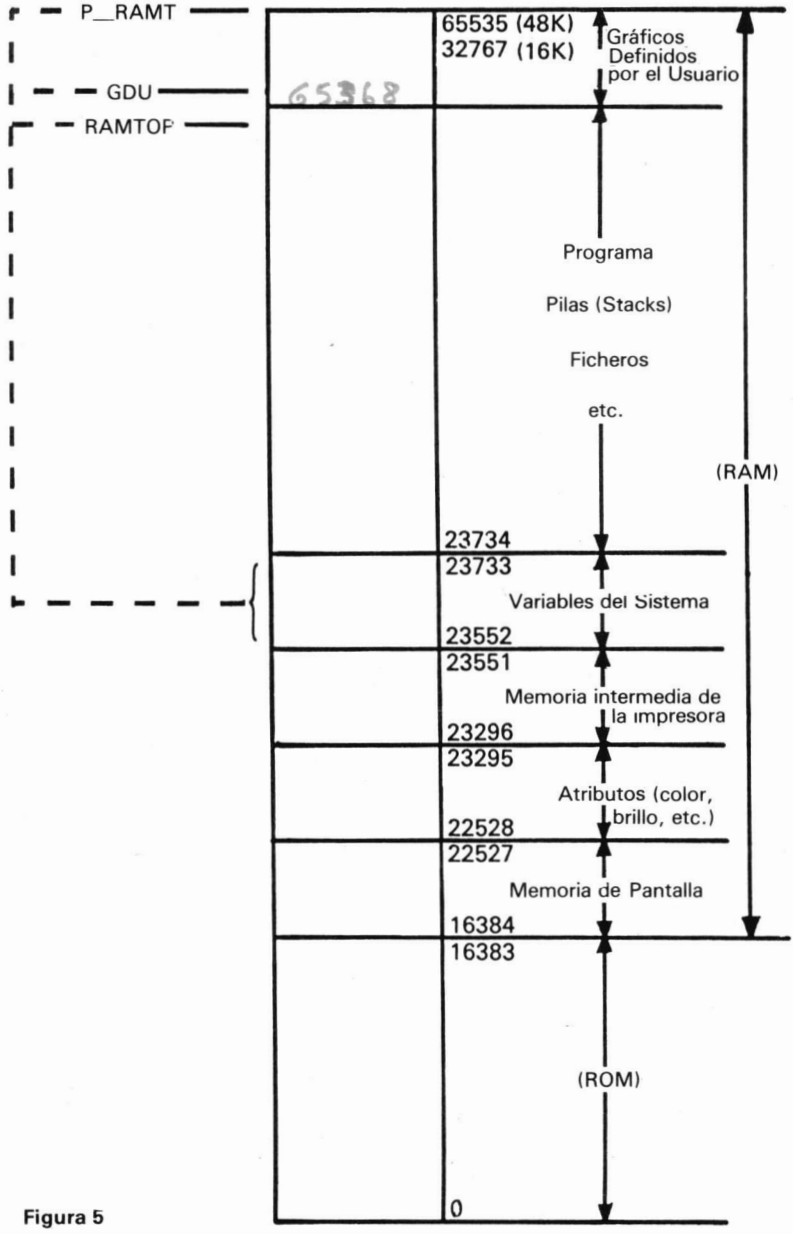

P_RAMT
GDU
RAMTOP
65535 (48K)
32767 (16K)
Gráficos Definidos por el Usuario
Programa
Pilas (Stacks)
Ficheros
etc.
(RAM)
23734
23733
Variables del Sistema
23552
23551
Memoria intermedia de la impresora
23296
23295
Atributos (color, brillo, etc.)
22528
22527
Memoria de Pantalla
16384
16383
(ROM)
0

Figura 5

Cada «vivienda» tiene asignada una dirección de acuerdo con el «piso» en el que se encuentra y su posición dentro de dicha planta. Si numeramos las plantas secuencialmente desde 0 hasta 255, y las viviendas (posiciones) también las numeramos desde 0 hasta 255 en cada una de las 256 plantas, resultará posible localizar cualquier vivienda especificando sus números de «planta» y «puerta», respectivamente. Por ejemplo: piso 10, puerta 54. La posición de esa dirección dentro del total de 65536 viviendas puede determinarse rápidamente como sigue:

(número del piso × 256) + número de la puerta.
(10 × 256) + 54 = 2614

De modo análogo, la posición más elevada sería (255 × 256) + + 255 = 65535. Esta es, en efecto, la forma en que el Spectrum establece el direccionamiento de sus 65536 posiciones de memoria y aunque se trate de un producto Sinclair (sin embargo, se trata de una función del Z80)— el ordenador se empeña en hacerlo «al revés». Utiliza dos bytes de memoria para manejar una dirección. El primero almacena la «parte menos significativa» de la dirección (54 en el ejemplo citado); y el segundo lo dedica a almacenar la «parte más significativa» (10 en el ejemplo indicado). Los códigos de dirección de 16 bits utilizados por la CPU del Z80 son dos bytes, de 8 bits, considerados como un solo número largo. Los primeros ocho bits constituyen el *byte menos significativo,* y los ocho últimos el *byte más significativo,* normalmente mencionados por las contracciones «LSB» y «MSB» («Least Significant Byte» y «Most Significant Byte», respectivamente).

Determinación de los bytes de memoria que ocupa un programa

Un buen ejemplo de lo dicho se puede ofrecer con una sencilla fórmula —de gran utilidad— que nos dice cuántos bytes de memoria han sido ocupados para almacenar un programa de BASIC. Teclea, o carga, cualquier programa en BASIC y luego entérate de los bytes ocupados haciendo lo siguiente:

```
PRINT (PEEK 23653 + (PEEK 23654) * 256) — (PEEK 23635
+ (PEEK 23636) * 256)
```

Esto lo que hace es, sencillamente, averiguar dónde comienza la memoria desocupada (STKEND) y se le resta la dirección inicial del programa escrito en BASIC. Estas direcciones se hallan PEEKeando las variables del sistema anteriormente señaladas. Cada dirección está almacenada en dos bytes de memoria, siendo el primero el LSB. La primera dirección se halla, por tanto, «mirando» (PEEK) la posición de memoria 23653, extrayendo su contenido (un byte de 8 bits) y sumándole el contenido de la posición de memoria siguiente multiplicada por 256. La segunda dirección se halla del mismo modo.

La función de «PEEK» y la de «POKE»

Al examinar las direcciones nos hemos encontrado con la expresión PEEK, la cual es totalmente descriptiva de la función que desempeña («peek» en inglés significa «ojeada»). Nos permite mirar dentro de una posición de memoria y ver lo que hay allí —como si miráramos a través de la ventana de una de las 65536 viviendas de nuestro edificio de apartamentos. Eso es lo único que podemos hacer en la parte de la memoria ocupada por la ROM —¡MIRAR pero no TOCAR!—. De todas formas, con respecto a la parte ocupada por la RAM podemos hacer ambas cosas: mirar y tocar. La orden o comando para «tocar» o cambiar es «POKE» y es el que nos permite cambiar el contenido de una posición de memoria POKEando un nuevo valor. ¡Pruébalo! Dile a tu Spectrum que escriba PRINT PEEK n, en donde «n» es la posición de la RAM que hayas elegido.

Una vez hayas averiguado el contenido de esa posición (escrita en la pantalla), entonces haz POKE n,x —en donde «n» es la misma posición anterior y «x» es un valor diferente elegido por ti entre 0 y 255. Después escribe PRINT PEEK n de nuevo y se confirmará que has cambiado, con éxito, el contenido de esa posición de memoria. A lo largo de este libro te encontrarás con que tendrás que hacer PEEK y POKE gran cantidad de veces.

Casi sin darnos cuenta nos estamos acercando al momento en que estaremos preparados para hablar al Spectrum en su propio idioma. Ya conocemos de qué están hechas las «palabras», donde son almacenadas y cómo, en principio, podemos llegar a ellas en el ordenador —¡las «pokeamos»! (las introducimos)—. ¿Necesitamos conocer algún concepto más? La verdad es que debemos ser capaces

de preparar al ordenador para recibir nuestras instrucciones y, por supuesto, será necesario conocer exactamente qué palabras pueden ser entendidas y procesadas. En principio, el microprocesador Z80 sólo podría entender 256 instrucciones de Código Máquina pero —de nuevo, no te sorprendas— puede manejar muchas más instrucciones a través de un uso adecuado de «prefijos» suplementarios. De hecho, puede reconocer más de 600 y ser ejecutadas por la CPU. Todas ellas están en las listas del Apéndice B. Vamos a ver ahora algunos principios fundamentales.

La ejecución de rutinas en Código Máquina

Las rutinas en Código Máquina siempre se pueden ejecutar a través de un programa BASIC utilizando el comando «RAND USR»*. Por eso, para tener acceso a la rutina en Código Máquina utiliza la Subrutina del Usuario (**U**ser **S**ub**R**outine), indicando el sitio de la memoria en el que se encuentra. Por ejemplo, RAND USR 30000 llamará a una rutina de la máquina cuyo primer byte está almacenado secuencialmente en la posición de memoria 30000. El resto de la rutina estará almacenado «de ahí para arriba». ¿Cómo se coloca la posición inicial en el primer lugar? Bien, lo único que debemos hacer es decidirnos y dar órdenes al ordenador. Introduciendo CLEAR 29999 (o cualquier otra posición adecuada) hace el mismo efecto que poner el RAMTOP (tope de la RAM o posición más elevada de la misma) en esa dirección y así, de esta manera, la identifica como la última posición en la que puede colocarse o insertarse un programa BASIC. Entonces, la rutina en Código Máquina empezará en la posición siguiente —30000.

Si utilizamos el comando NEW, éste borra la RAM solamente hasta el RAMTOP, y cualquier rutina en Código Máquina que hayamos almacenado por encima del RAMTOP está protegida, tanto de ser «renovada» como de ser «superpuesta» por el BASIC. Si utilizamos el comando RUN, éste «borrará» (hará un CLEAR) las variables pero, de ningún modo, cambiará el RAMTOP. Por consiguiente, si se utiliza CLEAR con una dirección de memoria concreta se

---

* *N. del T.:* «RAND» es la abreviatura de «RANDOMIZE». «USR» es la abreviatura de «User Sub-Routine».

puede «subir» el RAMTOP para dejar más espacio al BASIC (superponiéndose a los Gráficos Definidos por el Usuario), o «bajar» dicho tope para aumentar la zona de RAM que queda protegida del NEW.

Una vez que hemos decidido dónde va a ser almacenado el Código Máquina, ¿cómo se ejecuta? Como hemos dicho anteriormente, éste debe ser almacenado secuencialmente ya que ésta es la forma en que la CPU resuelve sus asuntos. El proceso de BUSQUEDA y EJECUCION comienza en la dirección estipulada y luego sigue su curso a través de las sucesivas posiciones de memoria, interpretando cada cosa que encuentra como una instrucción y, por eso, lo que hace es ejecutarla. Este proceso continúa hasta que se le manda parar —o hasta que «se vuelve loco». A continuación se ofrece un programa entretenido que nos demuestra esta forma «secuencial» de trabajar. Por favor, ¡no te preocupes de cómo lo hace! La cosa consiste en que la ROM del Spectrum ¡se ha vuelto loca!

```
  5 RESTORE
 10 READ N
 20 LET A=120
 25 PLOT 55,27: DRAW A,A,N*PI
 32 CLS
 33 IF N=9999 THEN  GO TO 5
 35 GO TO 10
 40 STOP
 50 DATA 597,631,315,343,297,63
1,613,787,313,741,187,147,279,99
99
```

Cuatro formas distintas de cargar el Código Máquina

El pasatiempo anterior nos permite observar cuatro formas distintas de CARGAR el Código Máquina.

1. Haciendo POKE de un byte cada vez.

```
POKE X,Y
```

En donde X es la dirección inicial e Y es el valor que hay que introducir. Se hace POKE del segundo byte será en la posición X + 1; del tercero en la posición X + 2, etc.

2. A través de un bucle FOR/NEXT.

```
10  FOR A=1 TO X
20 INPUT Y
30 POKE A+POSICION INICIAL,Y
40 NEXT A
```

3. Mediante la LECTURA (READ) de datos (DATA).

```
10  FOR A=1 TO X
20 READ N
30 POKE POSICION INICIAL+A,N
40 NEXT A
50 DATA N1.,N2,N3,N4
```

En donde N1, N2, N3, N4 ... son la secuencia de bytes de la rutina en Código Máquina.

4. Cargarlo (mediante LOAD) de una cinta de cassette.

   CLEAR posición elegida
   LOAD «nombre» CODE.

Las opciones 2 y 3 nos ofrecen dos fundamentos distintos para crear un Cargador de Código Máquina. Claramente queda descartada la «carga» de combinaciones binarias, y es aquí donde nuestros conocimientos recientemente adquiridos sobre los Hexadecimales pueden sernos de gran utilidad. ¡Atención!, un CARGADOR HEX.:

```
  5 CLEAR 29999
 10 PRINT "***CARGADOR DE CODIG
O MAQUINA***"
 20 PRINT "        © JAMES WALSH
1983"
 30 INPUT ''"DIRECCION INICIAL?
 ";DIRECCION
 40 PRINT '''"INTRODUCE UN CODI
GO CADA VEZ QUE"
 50 PRINT "SE TE PIDA. HAZLO CO
N MAYUSCULAS"''
 60 PRINT DIRECCION;" = ";
```

```
  70 INPUT A$
  80 IF A$="FIN" THEN  GO TO 100
0
  85 IF LEN A$<>2 THEN  GO TO 70
  90 LET ALTO=CODE A$(1)-48
 100 IF ALTO>9 THEN  LET ALTO=AL
TO-7
 110 LET TOTAL=16*ALTO
 120 LET BAJO=CODE A$(2)-48
 130 IF BAJO>9 THEN  LET BAJO=BA
JO-7
 140 LET TOTAL=TOTAL+BAJO
 150 PRINT A$
 160 POKE DIRECCION,TOTAL
 170 LET DIRECCION=DIRECION+1
 180 GO TO 60
1000 PRINT 'AT 21,0;"FIN DEL PRO
GRAMA"
2000 RANDOMIZE USR 30000: GO TO
2000
```

N. del T.: En la línea 5 del programa se ha incluido un CLEAR 29999 ya que la gran mayoría de las rutinas de Código Máquina del libro comienzan en la posición 30000. No obstante deberás cambiar dicha posición si deseas colocar la rutina de Código Máquina en una posición diferente.

Hay «registros» que no sirven para inscribir propiedades

Capítulo 4

OPERACIONES CON LOS REGISTROS

Una vez leídos los capítulos anteriores, ya tendrás una idea razonable del tipo de números que puede entender el ordenador, la forma en la que podemos utilizar esos números y el modo en que los maneja el ordenador para que podamos manipular la memoria del mismo; como si se tratara de una larga fila de cajas vacías —cada una de las cuales contiene unos números específicos con el objeto de tener acceso a esas cajas sin necesidad de mirar lo que hay a lo largo de todas ellas— para sumarles o restarles posteriormente alguna cantidad. Lo que haremos en este capítulo será ver la forma en que el ordenador controla estos números cuando se encuentran fuera de sus cajas, y cómo podemos manipularlos. Veremos la forma en que el ordenador guarda estos números, la forma en la que el ordenador guarda los comandos en Código Máquina, y también veremos alguna sencilla instrucción de Código Máquina relacionada con estas actividades. Luego empezaremos a ver cómo podemos utilizar esas instrucciones en nuestros propios programas. Al final dejaré que resuelvas un pequeño problema utilizando un programa en Código Máquina.

En Código Máquina hay registros en lugar de variables

Resulta muy interesante disponer de la posibilidad de poner números en una caja y poder retirarlos más adelante pero esto no

nos es de gran utilidad a no ser que luego nos sea posible hacer algo más que eso. Cuando escribes un programa en BASIC utilizas una variable. Por ejemplo, si quieres que sea almacenado un número como la variable A, simplemente introducirás LET A = el número. Luego podrás hacer lo que quieras con esa variable A. La podràs sumar a otro número, multiplicarla o restarla y, una vez que hayas terminado, podrás borrar esa variable y dejarla en cero con sólo escribir LET A = 0.

Desgraciadamente, en Código Máquina no hay variables. En su lugar tenemos algo a lo que llamamos REGISTROS. Hay siete registros de un byte que pueden ser fácilmente utilizados y manejados. Todavía hay más pero éstos son los que utiliza principalmente el ordenador. Naturalmente podemos utilizarlos pero con un poco más de dificultad y, por esta razón, es por la que dejaremos estos registros para más adelante. Durante la mayor parte del tiempo que dediques a la programación en Código Máquina, los únicos registros que vas a utilizar van a ser los siete «fáciles». Cada registro tiene una letra de identificación: A, B, C, D, E, H y L. No existe ninguna razón por la que se hayan elegido estas siete letras y no hay motivo para preocuparse tratando de buscar sentido a este convenio. Como ya sabes, el ordenador, cuando maneja un solo byte, sólo puede operar con números entre 0 y 255. Debido a que cada uno de estos registros es un REGISTRO DE UN BYTE, el número más grande que puede almacenarse en cualquiera de ellos es el 255. Sin duda ya te habrás dado cuenta de que es un valor demasiado bajo para que un número sea práctico (¿cuántos programas has hecho hasta ahora cuyos números no sobrepasen el 255?). Para superar esta limitación lo que hace el ordenador es agrupar dos registros de un byte para formar lo que se llama un REGISTRO DOBLE. Si se combinan dos registros de esta forma, el número más grande que pueden manejar es 256 multiplicado por 256, menos 1, lo que nos da un total de 65535. En terminología de ordenadores esta serie (rango) entre 0 y 65535 se representa por 64K. La razón por la que tenemos que restar 1 al final, se debe sencillamente al hecho de que la serie va desde cero a 65535, lo que significa que necesitamos un byte más para el número cero.

Los registros no pueden agruparse de cualquier forma. Las combinaciones fijas son: BC, DE, HL, tal y como se ve en la figura 6.

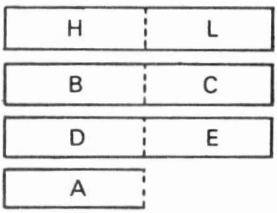


Figura 6

Si echas una rápida ojeada a la lista de los comandos del Código Máquina y a sus códigos Hex al final del libro, notarás que bajo el encabezamiento «MNEMONICOS» hay un gran número de comandos que están directamente relacionados con estos registros. Llegarás a darte cuenta de que el conocimiento completo de los registros y de su funcionamiento, te serán de mayor utilidad que cualquier otra cosa en programación con Código Máquina. Otra facilidad de la CPU del Z80 que, una vez más, nos demuestra ser inapreciable, es su posibilidad de utilizar los registros como registros de un byte, es decir, si utiliza uno lo hace entre 0 y 255, y si lo hace combinando dos, entre 0 y 65535. Esto forma parte de la capacidad interna del ordenador.

En breve pasaremos a ver cómo se introducen los números de estos registros y cómo sumar y restar con ellos. Más adelante también veremos la manera de realizar operaciones más complicadas con éstos y con otros registros. Mientras que otros registros nos serán de menor utilidad y otros están reservados para operaciones propias del ordenador, habrá ocasiones en las que el acceso a estos registros nos dará grandes ventajas en nuestros programas.

Probablemente la manera más fácil de penetrar en el concepto de «registro» es pensar que el ordenador es una persona que lleva un abrigo. Este abrigo cuenta con siete bolsillos, cada uno de los cuales tiene una etiqueta diferente. Al ordenador le resulta muy fácil sacar cosas de estos bolsillos y poner otras en su lugar. Para hacer esta operación es necesario «dirigir» al ordenador porque éste no tiene capacidad para pensar por sí mismo, y para ello debe ser instruido con precisión. El ordenador sólo puede entender aquellas órdenes que se le dan en forma de números o códigos.

El lenguaje ensamblador

El lenguaje en el que las órdenes —o comandos— son únicamente números se conoce como LENGUAJE MAQUINA. Esto es, en sí mismo, el fundamento del Código Máquina. Desgraciadamente, mientras que resulta bastante fácil introducir comandos en forma de números en la memoria del ordenador (utilizando el comando «POKE»), por el contrario, para un simple mortal ya no es tan fácil tratar de acordarse del código de un comando concreto. Pero aún no hay que desesperar porque, como ha sido un gran problema para otros, se ha logrado una sencilla solución. La respuesta a nuestro problema nos llega en forma de «LENGUAJE ENSAMBLADOR». El lenguaje ensamblador se compone de un juego de instrucciones que representan exactamente lo mismo que los códigos en lenguaje máquina, pero con la ventaja de que nos resultan más fácilmente comprensibles. Por ejemplo, una instrucción en Código Máquina se puede parecer a esto: 01101011, mientras que el equivalente en lenguaje ensamblador sería: LD A, 1. Lo de «más fácilmente comprensible» está referido a aquellas personas angloparlantes pero, ¡tranquilidad!, todo será explicado convenientemente para el lector hispanoparlante.

Los mnemónicos

Mientras que por ahora la cosa no parece tener mucho significado, lo que ya resulta evidente es que la forma habitual con la que dábamos sentido a los «agrupamientos» de letras y de dígitos, los cuales hemos utilizado hasta ahora, nos permite enfrentarnos más fácilmente con la versión del lenguaje ensamblador que con una cadena de dígitos binarios. Esto resulta todavía más importante si tratamos de aprender de memoria cualquiera de estos números. Estas representaciones «humanizadas» de los comandos binarios están convenientemente apodados «MNEMONICOS»*.

Desafortunadamente no nos es posible escribir los mnemónicos directamente en el ordenador —debido simplemente a que el ordenador no entiende nuestros mnemónicos— y, en su lugar, tenemos

---

* *N. DEL T.:* Utilizaremos esta traducción de la palabra «Mnemonics», en lugar de la más correcta «mnemotécnicos», por su mayor aproximación a dicha palabra inglesa y por lo extendido de su uso en el lenguaje de ordenadores.

que transformarlos de manera que puedan ser comprendidos por el pobrecito ordenador. De una manera u otra esto significa su reconversión en binarios. Hay dos métodos fundamentales para lograrlo. La forma lenta, pero perfectamente eficaz, de hacerlo es la de utilizar las tablas de conversión existentes al final del libro. Una más fácil y, por añadidura, más rápida, es la que consiste en utilizar un programa ENSAMBLADOR prefabricado.

A medida que vayamos avanzando a través del libro iré presentándote, por supuesto, cada uno de los comandos importantes junto con su mnemónico. Te explicaré su significado y su uso y, cuando tengas que trabajar con los ejemplos propuestos, podrás llevar a cabo la conversión de los «mnemónicos» en su «código», y luego introducirlos en el ordenador utilizando el «cargador» que se facilita. Descubrirás, sin duda, que un Ensamblador presta una gran ayuda. Sin embargo, no será necesario que salgas precipitadamente a comprar uno. Si decides que quieres acortar algo de trabajo y, de todos modos, deseas tener un Ensamblador disponible para cuando me dedique a él más adelante y con más detalle, entonces puedes ir buscando proveedores de Ensambladores que puedan suministrártelo.

Es importante recordar que los lenguajes ensambladores no son una adaptación del Código Máquina como lo es el BASIC. En lenguaje ensamblador sólo hay un mnemónico para cada instrucción de lenguaje máquina y, viceversa, sólo hay una instrucción en Código Máquina para cada mnemónico en lenguaje ensamblador. Podemos afirmar, además, que el lenguaje ensamblador es, en efecto, equivalente al lenguaje máquina. Casi todos los mnemónicos son abreviaturas de las operaciones que controla cada comando de lenguaje máquina. Por lo tanto, esto es, casi siempre, un modo muy sencillo de convertir las abreviaturas en instrucciones. Por ejemplo: INC HL es la abreviatura de «*inc*rementar HL». De modo análogo, LD A,0 es una abreviatura para «cargar A con un 0» («*load* A con un 0»). Los comandos equivalentes en lenguaje máquina de estas instrucciones son 23 y 62,0, respectivamente.

Las ventajas que ofrece un ensamblador

La gente hace listados de sus programas de modos diferentes, por ejemplo, en lenguaje máquina o en lenguaje ensamblador. Es

más fácil entender un programa si éste se encuentra en lenguaje ensamblador pero un ensamblador para el Spectrum resulta un tanto caro, aproximadamente entre cinco y nueve libras esterlinas (en España, independientemente del cambio de la libra, cuestan entre 1.500 y 2.500 pesetas). Por mucho que compares entre el dinero que cuesta y el tiempo y las energías que te ahorrarás tecleándolo, más lo que ganarás con la posibilidad de entender lo que estás escribiendo, el ensamblador resultará ser una gran ayuda. Si decides no utilizar un ensamblador o no tener uno por cualquier motivo, entonces todavía te será bastante fácil introducir los programas en el ordenador a través de un «cargador Hex». Para que esto sea posible me he preocupado de listar en lenguaje ensamblador y en lenguaje máquina todos los programas que aparecen en este libro y así, cualquiera que sea la forma en que hayas decidido introducir el programa, te será posible observar la versión en lenguaje ensamblador para saber qué es lo que haces.

Recuerda que, debido a que el Código Máquina está creado para una CPU determinada —como es el caso del Z80—, si eres capaz de escribir en Código Máquina para el Spectrum, no te costará mucho trabajo escribirlo para el ZX81, el ZX80 o incluso para cualquier otro ordenador que utilice la CPU del Z80. Conviene tenerlo presente a la hora de ir a comprar un ordenador, en el caso de que ya hayas tenido un Spectrum o si decides ¡sumarlo a tu equipo de microordenadores! Por ejemplo, podrías optar por comprar un ordenador BBC, o un Commodore VIC20, entonces tendrías que volver a aprender gran cantidad de Código Máquina para que te fuera posible utilizarlo en estos otros ordenadores pero con el ZX81 y con el Lynx tendrás muy poco que aprender antes de poder escribir los mismos buenos programas en Código Máquina para ellos. No se trata del único factor a la hora de decidir si vas o no vas a comprar un nuevo ordenador pero puede ser de utilidad que lo tengas en cuenta.

Un programa demostrativo de lo que es el Código Máquina

Bien, tomemos un pequeño descanso. He aquí un interesante programa que nos va a servir como ejemplo sencillo de lo que es realmente el Código Máquina. Por favor, fíjate en las dos secciones

de este programa: el listado del lenguaje ensamblador y el del lenguaje máquina. Este programa realiza el scroll, pixel por pixel, hacia la izquierda.

```
ORG 30000
30000 21 FF 57      LD HL,22527
30003 E5            PUSH HL
30004 11 20 5B      LD DE,23328
30007 01 20 00      LD BC,32
30010 ED B8         LDDR
30012 E1            POP HL
HL=DIRECCION MAS BAJA DEL BLOQUE
   INFERIOR
B=NUMERO DE BLOQUES
30013 06 03         LD B,3
A
30015 C5            PUSH BC
30016 06 08         LD B,8
B
30018 C5            PUSH BC
30019 E5            PUSH HL
30020 06 08         LD B,8
C
30022 C5            PUSH BC
30023 E5            PUSH HL


30024 E5            PUSH HL
30025 D1            POP DE
30026 25            DEC H
30027 01 20 00      LD BC,32
30030 ED B8         LDDR
30032 E1            POP HL
30033 25            DEC H
30034 C1            POP BC
30035 10 F1         DJNZ,C
30037 24            INC H
30038 11 E0 06      LD DE,1760
30041 E5            PUSH HL
30042 19            ADD HL,DE
30043 D1            POP DE
```

```
30044 01 20 00      LD BC,32
30047 D B8          LDDR
30049 E1            POP HL
30050 11 20 00      LD DE,32
30053 ED 52         SBC HL,DE
30055 C1            POP BC

30056 10 D8         D JNZ,B

30058 11 00 07      LD DE,1792
30061 ED 52         SBC HL,DE
30063 E5            PUSH HL
30064 11 20 00      LD DE,32
30067 19            ADD HL,DE
30068 E5            PUSH HL
30069 D1            POP DE
30070 E1            POP HL
30071 E5            PUSH HL
30072 01 20 00      LD BC,32
30075 ED B8         LDDR
30077 E1            POP HL
30078 C1            POP BC
30079 10 BE         DJNZ,A
30081 11 20 00      LD DE,32
30084 19            ADD HL,DE
30085 EB            EX DE,HL
30086 21 20 5B      LD HL,23328
30089 06 20         LD B,32

D
30091 7E            LD A,(HL)
30092 00            NOP
LD A,0 PARA EVITAR BUCLE SIN FIN
30093 12            LD (DE),A
30094 1B            DEC DE
30095 2B            DEC HL
30096 10 F9         DJNZ,D
30098 C9            RET
```

La introducción de números en los registros

Ahora ya estás familiarizado con la forma en la que los registros de un solo byte (a partir de ahora «Registros Sencillos») pueden combinarse con los Registros Dobles. Es probable que sea un buen momento para recordarte que los registros sencillos pueden manejar números entre 0 y 255, mientras que los registros dobles pueden operar con números entre 0 y 65535. Vamos a empezar viendo cómo introducir números en los registros sencillos (registros de un sólo byte).

Supongamos, por ejemplo, que quieres poner un número —digamos el 11— en uno de los registros (o «bolsillos»). Para hacerlo deberás meter ese número en el «bolsillo». Esto se hace con el comando llamado LD, el cual es la abreviatura de «LoaD» (introducir o cargar). Dado que hay una serie de registros o «bolsillos» diferentes, también tendrá que haber el mismo número de códigos de lenguaje máquina para las instrucciones, es decir, uno para cada una. Los códigos específicos de los mnemónicos de carga son los siguientes:

MNEMONICO	HEX	DECIMAL
LD A, xx	3Exx	62,xx
LD B, xx	06xx	6,xx
LD C, xx	0Exx	14,xx
LD D, xx	16xx	22,xx
LD E, xx	1Exx	30,xx
LD H, xx	26xx	38,xx
LD L, xx	2Exx	46,xx

La primera parte de este proceso consiste en identificar en qué registro se va a introducir. En la segunda parte hay que concretar el número que vamos a introducir en el registro elegido. Esto significa escoger el A, el B, el C (u otro registro) y concretar el número que se va a introducir. La representación para introducir el 11 en el registro A con el lenguaje ensamblador tiene este aspecto: LD A,0B. Sin embargo, antes de que pueda ser ejecutado, necesitaríamos un programa ensamblador con objeto de que el ordenador pueda entender lo que queramos que haga. Ahora, si estás utilizando un cargador Hex —lo cual te recomiendo por ahora, así entenderás lo que va

sucediendo—, es necesario introducir primero el número 3E. Esto le dice al ordenador que va a ser introducido un número en el registro A. A continuación pon el número 0B y así entenderá que éste es el número que vamos a introducir en el Registro A.

> *Nota:* Es fundamental recordar que cuando el código de un comando o un comando de lenguaje ensamblador es seguido de dos o de cuatro «Xs», significa que has de poner uno o dos bytes. Por ejemplo, cuando utilizamos el comando LD A,XX, las dos «X» significan que debemos poner un número de un byte (dos dígitos Hex) después del comando LD A, para introducirlo en el registro A.

Igualmente y con igual sencillez podemos introducir un número en cualquiera de los registros A, B, C, D, E ..., pero tienes que recordar la necesidad de utilizar un código o un mnemónico de lenguaje ensamblador diferente para cada registro. (Véase texto anterior.) Antes de continuar, vale la pena que veamos cómo funciona esto. Hay dos cosas importantes que debemos meternos en la cabeza (aunque por ahora no sea necesario que las entendamos), antes de que podamos crear y utilizar un programa en Código Máquina. La primera es que, al final del programa en Código Máquina, es necesario poner el código del comando RETurn (regreso). Este RETurn actúa exactamente de la misma manera que cuando utilizamos un GOSUB (ir a subrutina) en una rutina de BASIC. Por ejemplo. si haces GOSUB 1000 en un programa BASIC, y llegas al final de esa subrutina que comienza en el 1000, pondrás el comando RETURN al final de la misma para decirle al ordenador que deseas regresar al comando inmediatamente después de la instrucción GOSUB 1000. Así, si accedemos desde el BASIC a una rutina en Código Máquina por medio del comando USR del BASIC, que significa **U**ser **S**ub-**R**outine (SubRutina del Usuario) nos será preciso introducir una instrucción RET al final de la rutina en Código Máquina. De esta forma regresará al programa e irá al siguiente comando BASIC disponible. En breve veremos el porqué de todo esto pero, por ahora, lo que importa es que te des cuenta de ello. El código en lenguaje máquina para RETurn es C9.

La segunda cosa a tener en cuenta es que si accedes a un programa en lenguaje máquina a través del comando PRINT USRxxxxxx,

entonces el número que hay que escribir al finalizar la subrutina del usuario, es el contenido en el registro doble BC. Este detalle podemos utilizarlo ventajosamente.

Lo único que hace el programa que se presenta a continuación es cargar los registros B y C con el 0 vuelve (RETurns) al BASIC, para escribir un 0. Para cargar este programa utiliza el programa cargador de Código Máquina, o el programa cargador Hex que se facilita, y teclea simplemente los números Hex que se ofrecen. Una vez que el programa haya sido introducido en la memoria del ordenador, no trates de ejecutar con RUN el programa. Unicamente escribe «FIN» e introduce directamente el comando BASIC «PRINT USR 30000». De esta forma, la rutina en Código Máquina regresará al BASIC pero, al mismo tiempo, escribirá el contenido de BC. Una vez hecho esto, prueba a cambiar el programa poniendo números diferentes en los registros B y C, y averigua cuál será la respuesta que aparecerá en la pantalla. Recuerda que estás trabajando con un número de dos bytes con un máximo de 255 decimal en cada byte.

MNEMONICO	HEX
LD B,00	0600
LD C,00	0E00
RET	C9

El proceso más lógico sería el de cargar el contenido de un registro dentro de otro con el mismo procedimiento con que cargábamos un número dentro de un registro. Para ello hemos de tener diferentes códigos de lenguaje máquina con cada una de estas operaciones (por ejemplo, cargar el registro B en el registro A), dando lugar a un gran número de códigos distintos y, por este motivo, debemos concentrarnos en cargar los distintos registros en el registro A para facilitar la explicación y su comprensión. El principio es el mismo para todos los demás registros, excepto que los códigos en lenguaje máquina y los mnemónicos son diferentes. Por lo tanto, entender uno significa entenderlos todos.

Si quisieras cargar el contenido del registro B en el registro A entonces lo podrías conseguir mediante el comando sencillo LD A,B (que significa «Cargar A con B»). Si acudimos de nuevo a nuestra

anterior analogía de que el ordenador es como una persona que lleva un abrigo, podemos pensar que esta operación se ha realizado así:

1. Se mira en el bolsillo A, se extrae su contenido y nos deshacemos de él.
2. Se mira en el bolsillo B y se averigua cuántos artículos hay en el bolsillo B pero sin tocarlos.
3. Se toma el mismo número de artículos que hubiere en B y se colocan en el bolsillo A.

Ahora hay el mismo número de artículos en el bolsillo A y en el bolsillo B. Es importante advertir que el valor inicial del contenido de A no tiene ninguna relación con el resultado final de la operación.

Puedes cargar el contenido de cualquier otro registro en el registro A. Los mnemónicos y los códigos hexadecimales del lenguaje máquina de estas instrucciones son los siguientes:

MNEMONICOS	HEX	DECIMAL
LD A,A	7F	127
LD A,B	78	120
LD A,C	79	121
LD A,D	7A	122
LD A,E	7B	123
LD A,H	7C	124
LD A,L	7D	125

Lo que vamos a ver a continuación serán los mnemónicos y los códigos del Lenguaje Máquina para las instrucciones con las que cargar otros registros dentro de los registros BC, y utilizar esto de forma que la respuesta sea presentada en la pantalla al regresar al BASIC.

¡Tampoco nos ayudaría una caja registradora!

Capítulo 5

CÓMO REALIZAR SUMAS CON LOS REGISTROS

Ahora ya sabemos qué son los registros y cómo poner números en su interior, bien cargándolos directamente bien cargando un registro con otro, es necesario conocer cómo podemos manipular dichos registros. Si alguna vez has hecho programas en BASIC, sabrás lo inútil que sería asignar un número a una variable y no poder cambiarlo aritméticamente. Digamos que, para trabajar con un número, es necesario que podamos sumarle o restarle algo. Esto es lo que veremos en este capítulo. También veremos qué sucede cuando un número resulta demasiado grande o demasiado pequeño para ser guardado en un registro. Todo ello nos llevará a explorar el «SEÑALIZADOR DE ACARREO» («CARRY FLAG») y las piezas de información asociadas guardadas en el ordenador.

Ya habrás empezado a pensar acerca de cómo vamos a sumar dos registros y si nos va a ser posible o no hacerlo exactamente de la misma forma que lo hacemos en BASIC, en donde para sumar el contenido de la variable XY a la variable AB sólo hay que escribir: LET AB = AB + XY. La respuesta será guardada en AB. Afortunadamente podemos realizar una operación más sencilla en Código Máquina, excepto que dicha instrucción es totalmente diferente.

La suma de registros dobles

Si queremos sumar el contenido del registro doble DE al contenido del registro doble HL, en lugar de escribir LET HL = HL + DE

(que es lo que harías en BASIC), sólo es necesario utilizar ADD HL, DE. El ejemplo que incluimos nos enseña exactamente lo que pasaría si pusiéramos esta instrucción en un programa. Es interesante fijarse en que las dos primeras líneas contienen los dos números que van a ser sumados; la tercera línea tiene la instrucción que realiza la operación, y las líneas 4 y 5 cargan el resultado en el registro BC. Esto último ha de realizarse utilizando dos comandos separados —que suman a los dos registros sencillos— ya que no hay una instrucción del tipo LD BC, HL, y el último comando, que es el RETurn, sirve para regresar al BASIC. Te recomiendo que pruebes la ejecución de este programa:

```
ORG 30000
30000 21 01 00        LD HL,0001
30003 11 00 80        LD DE,32768
30006 19              ADD HL,DE
30007 44              LD B,H
30008 4D              LD C,L
30009 C9              RET
```

Por suerte para nosotros, el número máximo que puede guardar un registro doble es mayor que 65000, por lo que podemos presumir, con bastante seguridad, que la suma de dos números no va a sobrepasar este límite. De todas formas resulta interesante saber qué pasaría si esto ocurriera. El pequeño programa que se da a continuación nos lo aclarará. Pruébalo y averigua la respuesta.

```
ORG 30000
30000 21 01 00        LD HL,0001
30003 11 FF FF        LD DE,65535
30006 19              ADD HL,DE
30007 44              LD B,H
30008 4D              LD C,L
30009 C9              RET
```

La carga de un número en un registro doble

Antes de decirte la respuesta vamos a analizar rápidamente el ejemplo. El primer comando carga el registro doble HL con un 1.

«Sí» —ya te oigo gritar— pero el número después del 21 en el MNEMONICO no es el 1, es el 01 00. Ahora surge un difícil problema al que debes enfrentarte antes de seguir adelante. Cuando cargamos un número en un registro doble, en lugar de poner el byte más alto antes del byte más bajo o menos significativo, tenemos que hacerlo al revés. De aquí que para cargar el registro HL con un 1 es necesario teclear 01 00. El byte más alto o más significativo (00) va después del byte más bajo (uno). Si todavía no estás lo suficientemente seguro del significado de «el byte más alto» y el «byte más bajo», te valdrá la pena volver a ver la sección anterior del libro en donde se explicó por primera vez. Para reforzar tus conocimientos he aquí unos cuantos ejemplos rápidos de rutinas cortitas que utilizan este método. Afortunadamente no es tan raro cargar el byte más alto después del byte más bajo en un programa en Código Máquina, y no tardarás en acostumbrarte a ello. Prueba ahora esta breve rutina.

```
①       ORG 30000
        30000 01 00 01        LD BC,0100
        30003 C9              RET
```

La respuesta que deseamos es 1 pero, en la primera instrucción, hemos cargado el registro doble BC con los bytes alto y bajo «al revés». Por esta razón, la respuesta, en su lugar, nos dará 256.

Cuando escribimos en lenguaje ensamblador es habitual poner el número en hexadecimal y en el orden correcto, es decir, con el byte más alto antes que el byte más bajo. Como puedes ver en este primer ejemplo, al convertirlo al código hexadecimal, el byte más alto se ha vuelto a poner antes del byte más bajo. Esto es INCORRECTO.

```
②       ORG 30000
        30000 01 01 00        LD BC,0001
        30003 C9              RET
```

Esta vez el byte más alto o más significativo se ha colocado después del byte más bajo o menos significativo en el listado de código hexadecimal, de aquí que la respuesta correcta sea un 1.

③

```
ORG 30000
30000 01 FA 7B      LD BC,64123
30003 C9            RET
```

De nuevo, en este ejemplo vemos que el byte más alto y el byte más bajo han sido colocados al revés. Por consiguiente, la respuesta es completamente incorrecta. Ahora, antes de ejecutar la rutina siguiente, la número 4, prueba y averigua cuál sería la respuesta en decimal.

④

```
ORG 30000
30000 01 7B FA      LD BC,64123
30003 C9            RET
```

Ahora y rápidamente, para resumir y antes de volver con el ejemplo que veíamos anteriormente, podemos decir que: cuando cargamos un número en un registro doble, el byte más alto va después del byte más bajo cuando lo codificamos en lenguaje máquina hexadecimal. Este es otro ejemplo de cómo nos puede ayudar un ensamblador, en el sentido de que basta introducir el número con su configuración normal (es decir, configuración de byte más bajo).

En Código Máquina no hay códigos de error

Ahora volvamos con el ejemplo anterior. Ya hemos establecido que el primer comando carga el registro doble HL con un uno. El comando siguiente carga el registro doble DE con el FFFF en Hex, lo cual debes recordar, es equivalente a 65535 en decimal (que se trata del mayor número que se puede almacenar en un registro doble). Si ahora añadimos un 1 al mayor número almacenable, estaremos en una «situación de desbordamiento» o de exceso de capacidad (overflow). En BASIC, si se produce un desbordamiento, aparece un mensaje de error y el programa se detiene. En Código Máquina esto no ocurre porque no hay códigos de error. Sencillamente ¡no funciona! En este caso concreto lo único que pasa es que, en lugar de llegar a 65536, lo cual está fuera del margen, se vuelve a cero. En el caso de que se produzca desbordamiento el ordenador tam-

bién recuerda este hecho poniendo el «señalizador de acarreo» (carry flag) en 1. Por ahora no te preocupes de lo que es el señalizador de acarreo, pronto estaremos con ello.

Hay que recordar dos cosas importantes sobre la suma de dos registros dobles. Primeramente sólo al registro HL puede sumársele otro registro doble. Segundo que «sólo» son esos registros dobles los que pueden sumarse entre sí. Un registro sencillo (de un solo byte) no puede ser sumado a un registro doble y viceversa.

Cuando el Código Máquina exige diez veces menos memoria que el BASIC

A continuación se incluye una lista de los códigos hexadecimales de las operaciones de suma de un registro doble a otro registro doble. Te preguntarás por qué es necesario tener un comando que suma HL a sí mismo. La razón es muy sencilla ya que es una forma de doblar el valor de HL. También es interesante advertir que cada uno de estos comandos solamente ocupa un byte. Si comparas esto con su equivalente en BASIC verás rápidamente la diferencia: el equivalente en BASIC ocupa una cantidad de memoria ¡diez veces mayor!

```
ORG 30000
30000 09          LD HL,BC
30001 19          LD HL,DE
30002 29          LD HL,HL
```

La suma de registros sencillos

Vamos a ver ahora la suma de registros sencillos. Si recordamos que los registros sencillos sólo pueden almacenar números entre 0 y 255, la posibilidad de «desbordamiento» (overflow) es mucho mayor que la de los registros dobles. ¿Qué sucede cuando éstos se desbordan? Sencillamente se ponen a CERO al igual que los registros dobles. La pregunta es: ¿nos vale esto para algo? De hecho ¡sí! Siempre que sumemos dos números podrá haber, o no, un «desbordamiento» (o acarreo). El propio ordenador utiliza aparte un re-

gistro muy especial en el que anota cosas tales como los desbordamientos. Es conocido como el registro F. Para nosotros nos resultará excepcional el uso del registro F debido a que está monopolizado por el propio ordenador con objeto de manejar pequeñas cantidades de información. La forma de conseguirlo es clasificando cada bit de los demás para permitirnos utilizarlos separadamente. Si continúas leyendo llegarás a ver la alusión a un «señalizador» que ha sido «activado» («set») o «puesto a cero» («reset»). Esto referido a un bit significa que puede estar «marcado», que está «puesto a 1» —o «puesto a cero»— que «vale 0». Esto es a lo que llamamos SEÑALIZADOR DE ACARREO (CARRY FLAG). La mayor utilidad que nos ofrece el señalizador de acarreo es que, cuando se suman dos registros sencillos y la suma es mayor que 255, entonces el señalizador de acarreo será marcado (con un 1). Y, por supuesto, si la suma no supera 255, el señalizador de acarreo estará puesto a cero (con un 0). No podemos acceder directamente al registro F por lo que no nos es posible cambiar los señalizadores directamente, aunque nos es relativamente sencillo llegar a estos bits.

Es posible sumar al registro A cualquier otro registro, incluyendo al mismo registro A (aunque tampoco podrás sumar un registro a otro que no sea el A). A lo largo de tu aprendizaje te encontrarás con que el registro A, el cual a menudo se le llama el Acumulador, es utilizado muy frecuentemente. Para ello no hay ninguna razón particular, excepto que se trata de la primera letra del alfabeto —y la primera letra del Acumulador— y es una letra fácil de recordar. Muchos de los comandos más importantes utilizan este registro, así como, con los registros dobles, el registro HL es el más utilizado. Y del mismo modo que podemos sumar un registro al registro A, también nos es posible sumar un número al registro A, pero recordando que este número debe encontrarse entre 0 y 255. A continuación se incluye la lista de los distintos comandos ADD (suma) y sus códigos equivalentes de lenguaje máquina. Estos números forman una sucesión. De ahí que sea bastante fácil acordarse de ellos.

MNEMONICOS	HEX	DECIMAL
ADD A, (HL)	86	134
ADD A, A	87	135
ADD A, B	80	128

ADD A, C	81	129
ADD A, D	82	130
ADD A, E	83	131
ADD A, H	84	132
ADD A, L	85	133
ADD A, xx	C6xx	198, XX

La suma de registros sencillos es muy similar a la suma de registros dobles, tal como vemos en los ejemplos siguientes:

Ejemplo 1:

```
ORG 30000
30000 3E 05        LD A,5
30002 06 06        LD B,6
30004 80           ADD A,B
30005 4F           LD C,A
30006 06 00        LD B,0
30008 C9           RET
```

Ejemplo 2:

```
ORG 30000
30000 3E 0E        LD A,14
30002 C6 06        ADD A,06
30004 4F           LD C,A
30005 06 00        LD B,0
30007 C9           RET
```

En el ejemplo 1 las dos primeras instrucciones simplemente cargan un 5 y un 6 en los registros A y B respectivamente. El comando 3 suma el contenido del registro B al contenido del registro A y el resultado permanece en el registro A. Las dos líneas siguientes hacen que nos sea posible visualizar la respuesta utilizando el comando de BASIC "PRINT USR" (en este caso "PRINT USR 30000").

En el segundo ejemplo, simplemente sumamos 6 al contenido del acumulador o registro A antes de cargar el contenido de A en el registro C, cargar B con 0 y regresar (RET), de forma que el número exhibido después del comando PRINT USR sea el contenido en BC —o, con otras palabras—, la respuesta a A + 6. Una instrucción

ADD (suma) siempre afectará al señalizador de acarreo. Como se ha dicho anteriormente, si no hay "acarreo" el SEÑALIZADOR marcará un 0. Si hay "acarreo" marcará un 1.

La suma con acarreo

Hasta ahora no nos ha sido posible utilizar este mecanismo para nada. El modo más sencillo de utilizarlo es a través del comando ADC. Este comando significa "ADD with CARRY" (SUMA con ACARREO) y así es como funciona. Supongamos que el ordenador realiza una instrucción ADC A, B. Tomará el contenido del registro B, lo sumará al contenido del registro A, y dejará el resultado en el registro A al igual que sucedía con la anterior instrucción ADD A, B. Pero también sumará el señalizador de acarreo a este nuevo número. Como el señalizador de acarreo todavía no ha sido afectado por esta instrucción, el número que se le añade al resultado corresponderá a la posición anterior que tenía el señalizador. Una vez realizada esta operación almacena el resultado en el registro A, y reasigna el señalizador de acarreo. De ahí que, si el señalizador de acarreo ya estuviera marcado (puesto a 1) por una suma anterior del programa, y no hubiera sido cambiado desde entonces, el resultado del comando ADC resultará afectado por ello. A primera vista, esto puede parecer más un obstáculo que una ayuda para la programación. Sin embargo, si te acuerdas de los primeros días de escuela cuando aprendiste, por primera vez, a sumar, resultará evidente que se trata de la cosa más lógica y más útil. Sencillamente representa el principio de "llevar una" de la suma. Nos acordamos de que primero sumamos la columna de la derecha y, si hay un "desbordamiento", entonces debemos sumar un 1 a la siguiente columna de la izquierda. Por ejemplo, si sumamos 14 y 7, primero sumaremos el 4 y el 7, el resultado será 11, por lo que pondremos un uno debajo de la columna de la derecha y "llevamos una". Luego sumamos la columna de la izquierda en la que tenemos un uno y un cero y les sumaremos el número correspondiente al acarreo —en este ejemplo un 1— de ahí que el resultado sea 2 y éste se coloque debajo de la columna de la izquierda. Ahora ya podemos leer el resultado de izquierda a derecha leyendo 21 ó ventiuno.

El ejemplo siguiente nos muestra cómo podemos utilizar esta función en un programa en Codigo Máquina. Ya que el programa es un poco largo, y algo complicado, no te preocupes acerca de su explicación hasta que lo hayas introducido y hayas visto cómo funciona. Ya te lo explicaré en el párrafo siguiente. Por ahora asegúrate de que lo que introduces es exactamente lo que ves escrito en esta página.

```
ORG 30000
30000 16 33        LD D,51
30002 1E 85        LD E,133
30004 26 7B        LD H,123
30006 2E C7        LD L,199
30008 7D           LD A,L
30009 83           ADD A,E
30010 6F           LD L,A
30011 7C           LD A,H
30012 8A           ADC A,D
30013 67           LD H,A
30014 44           LD B,H
30015 4D           LD C,L
30016 C9           RET
```

El comando 1 asigna el valor 51 al registro D. El comando 2 asigna el valor 133 al registro E. El comando 3 asigna el valor 123 al registro H y el comando 4 asigna el valor 199 al registro L. El objeto de este programa consiste en sumar el contenido del registro doble DE al registro doble HL. Esto no se realiza a través de un sólo comando sencillo sino sumando las dos mitades de cada registro doble. En resumidas cuentas, lo que queremos hacer es, primero, sumar E a L y luego sumar el registro D al registro H. El primer problema con que nos encontramos es que sólo podemos sumar un registro al registro A, de ahí que es necesario sustituir al registro L por el registro A. Esto es bastante fácil y se realiza mediante el comando 5, el cual carga A con el contenido del registro L. Una vez hecho esto ya podemos sumar E al registro A. Por lo tanto, ahora el valor de A contiene la suma del registro L más el registro E pero no queremos que el resultado permanezca en A sino en L. Resolvemos este problema volviendo a almacenar el valor de A en L.

Debido a que ya hemos sumado los bytes menos significativos mediante la instrucción 6, el acarreo ya estaría marcado (con un 1) si hubiera habido un desbordamiento —que lo hubo—. Continuamos ahora casi con el mismo procedimiento para sumar los bytes más significativos, es decir, sumar el byte D al byte H. De todas formas, esta vez utilizamos el comando ADC —suma con acarreo—. Lo que sucede es que sumamos D al registro A, tal y como hicimos anteriormente, pero esta vez también le sumamos el contenido del señalizador de acarreo. Así, si cuando sumamos L y E el resultado fue mayor que 255, ahora tendríamos que añadir un 1 de más al valor de los bytes más altos. De esta forma podremos obtener el resultado exacto. Las instrucciones 11 y 12 son nuestros comandos preferidos los cuales nos permiten cargar el contenido de HL en BC con el objeto de ser presentados en la pantalla después de PRINT USR (30000) en BASIC. Otra cosa de este programa en la que debemos fijarnos es que los dos comandos que están después de ADD A, E, y antes de ADC A, D no afectan al contenido del señalizador de acarreo. Si vas a utilizar este tipo de instrucciones es útil saber qué comandos modifican el contenido del señalizador de acarreo para, de esta manera, saber qué resultado te vas a encontrar. Por esta razón, al final del libro, encontrarás una lista con todos los comandos señalando si afectan o no al señalizador de acarreo.

A estas alturas ya debes ser capaz de entender la diferencia entre ADD (suma) y ADC (suma con acarreo). Estos son los códigos y sus mnemónicos de las distintas combinaciones.

MNEMONICO	HEX	DECIMAL
ADC A, (HL)	8E	142
ADC A, A	8F	143
ADC A, B	88	136
ADC A, C	89	137
ADC A, D	8A	138
ADC A, E	8B	139
ADC A, H	8C	140
ADC A, L	8D	141
ADC A, xx	CExx	206, XX
ADC HL, BC	ED4A	237, 74
ADC HL, DE	ED5A	237, 90
ADC HL, HL	ED6A	237, 106

No es posible sumar directamente una constante a un registro doble pero se puede conseguir fácilmente cargando un registro doble con el número que desees sumarle a HL y luego sumar dicho registro doble a HL. El resultado en HL será igual a HL más ese número. Por ejemplo:

```
ORG 30000
30000 11 39 00        LD DE,57
30003 19              ADD HL,DE
```

Este método tiene la desventaja de que requiere la utilización del registro doble DE que, a lo mejor, quieres dedicar a otras operaciones. Otra forma de conseguir el mismo objetivo se muestra más adelante pero esta vez el único registro que es alterado, aparte del HL, es el registro A. Fíjate que en este ejemplo he vuelto a emplear el comando de "suma con acarreo" (ADC) de la misma forma que lo habíamos utilizado en el primer ejemplo, de modo que si el byte más bajo se desborda se acarreará un 1 al byte más alto, con lo cual tendremos el resultado con más precisión.

```
ORG 30000
30000 7D              LD A,L
30001 C6 39           ADD A,57
30003 6F              LD L,A
30004 7C              LD A,H
30005 CE 00           ADC A,0
30007 57              LD H,A
```

Me imagino que ya habrás tenido bastante trabajo de aprendizaje de Código Máquina por ahora. Por eso he aquí un programa en Código Máquina para que lo escribas y que lo utilices. Aunque tiene algunos comandos que ya hemos visto, no es necesario que a estas alturas seas capaz de entender el programa —o sea que no te enfades ni te preocupes si todavía queda algo de misterio— aunque al terminar el libro desearás volver a verlo y descubrir cómo funciona. La idea de poner aquí este programa es con el objeto de que lo utilices, veas el efecto de esta pieza de Código Máquina y tengas un pequeño descanso en el aprendizaje.

```
ORG 30000
30000 21 00 40        LD HL,16384
30003 06 C0           LD B,192
PARA DIFERENTES BLOQUES DE
PANTALLA CAMBIAR 'B' Y 'HL'
A
30005 0E 20           LD C,32
B
30007 5E              LD E,(HL)
30008 CB 1E           RR (HL)
30010 23              INC HL
30011 0D              DEC C
30012 20 F9           JR NZ,B
30014 CB 43           BIT 0,E
30016 28 09           JR Z,C
30018 E5              PUSH HL
30019 11 1F 00        LD DE,31
30022 ED 52           SBC HL,DE
30024 CB FE           SET 7,(HL)
RES 7,(HL) PARA EVITAR BUCLE SIN
 FIN
30026 E1              POP HL
C
30027 A7              AND A
30028 10 E7           DJNZ,A
30030 C9              RET
```

Como preludio de la siguiente sección del libro, mira este listado y fíjate en aquellos comandos que contienen un paréntesis formando parte de los mismos.

Como cargar el contenido de una posición de memoria en un registro

Anteriormente, como recordarás, hemos visto cómo se podían almacenar números dentro de cajas —o posiciones de la memoria. Y como también recordarás, cada una de estas cajas tenía un número que constituía la específica y concreta dirección de la caja. La razón de la existencia de esta dirección es que, de esta forma, podemos acceder al contenido de esta caja con relativa facilidad. Ya he seña-

lado lo útil que sería la posibilidad de manipular el número de las cajas pero, hasta ahora, todavía no hemos hablado de cómo quitar dichos números de sus cajas, o posiciones de memoria, y ponerlas en un registro. Por descontado, cuando se encuentran en un registro, nos será posible manipular los números adecuadamente. Luego también nos sería posible poner el contenido del acumulador, o de cualquier otro registro, dentro de una posición o dirección de memoria cualquiera. De esta forma, nos es posible guardar un número concreto para su utilización futura. También, usando este método, es posible tener una gran cantidad de números, todos ellos accesibles de una sola vez.

Vamos a considerar qué es lo que queremos hacer. Nuestro objetivo es introducir, digamos en el acumulador o registro A, el contenido de una posición concreta. Esta posición ha de tener una dirección. Si escribiéramos sencillamente LD A, con la dirección, entonces el ordenador confundiría la dirección con un número de acceso directo. Por ejemplo, si hubiéramos escrito LD A, 12, entonces sería razonable esperar que el ordenador considerara que queríamos cargar el acumulador con el valor 12. Pero lo que queremos es cargar el acumulador con el contenido de una posición concreta. De hecho esto es bastante sencillo ya que todo lo que tenemos que hacer es, en lugar de escribir el número después de la instrucción, colocamos el número entre paréntesis después de la "coma". Por ejemplo: LD A, (2465). Ahora el ordenador cargará el acumulador con el valor existente en la posición 2465. Debido a que cada posición sólo puede guardar un número entre 0 y 255 no hay problema en cargar un registro de un sólo byte (registro sencillo) con el contenido de una posición de memoria. Si dijéramos que en la posición 2465 estaba el número 64 entonces, después de ejecutar este comando, el acumulador también tendría el valor 64. Con todos estos comandos el contenido de la dirección todavía se mantiene. En otras palabras, el contenido de la dirección 2465 todavía es 64. Este es un comando muy útil, ya que ahora nos es posible tomar el contenido de cualquier posición de la memoria, ponerlo en un registro y manipularlo. Por ejemplo:

```
ORG 30000
30000 3A 30 75        LD A,(30000)
30003 C6 29           ADD A,41
```

El único problema es el de devolver ese número manipulado a la misma dirección, o a otra dirección. Esto puede conseguirse fácilmente utilizando una pequeña variación del comando anterior. Si piensas acerca de lo que queremos en este momento, te darás cuenta de que, en lugar de cargar el acumulador con el contenido de esa dirección, lo que deseamos ahora es poner en la dirección el contenido del acumulador. Lo que hacemos es utilizar el comando al revés: LD (DIRECCION), A sabiendo que la dirección la colocaremos entre los paréntesis. (ADDR = DIRECCION).

He aquí un sencillo ejemplo de cómo funciona este concepto.

```
ORG 30000
30000 3A 30 75      LD A,(30000)
30003 C6 0F         ADD A,15
30005 32 30 75      LD (30000),A
30008 C9            RET
```

Volviendo al primer concepto que utilizamos, es decir, el de tomar un número de una posición, manipularlo y luego devolverlo a la misma posición, podemos cambiar, por ejemplo, el color de la pantalla, cogiendo el color ya existente y manipulándolo de forma que consigamos otro color. Debido a que la pantalla, por lo que se refiere al color, tiene una longitud de 704 bytes, es necesario realizar esto 704 veces. Afortunadamente es posible poner en funcionamiento otro tipo de instrucciones que nos evitan escribir el programa ¡704 veces! Esto lo descubriremos más adelante. Ahora y antes de continuar con el ejemplo del programa del cambio de color que aparece más abajo, he aquí una rápida tarea para tí: escribe una rutina que coja el código de carácter de un carácter cualquiera de la memoria, cambiarlo a su inverso y devolverlo a su sitio. He aquí una pequeña información: el inverso de cada caracter es 128 veces mayor que el normal (en decimal). En otras palabras, si tomas el código del caracter "a" y le sumas 128, tendrás el código del "inverso de a".

```
ORG 30000
30000 21 00 58      LD HL,22528
30003 01 C0 02      LD BC,704
30006 1E            LD E,6
LOOP
```

```
30008 7E          LD A,(HL)
30009 CB 87       RES 0,A
30011 CB 8F       RES 1,A
30013 CB 97       RES 2,A
30015 83          ADD A,E
30016 77          LD (HL),A
30017 23          INC HL
30018 0B          DEC BC
30019 79          LD A,C
30020 FE 00       CP 0
30022 20 F0       JR NZ,LOOP
30024 78          LD A,B
30025 FE 00       CP 0
30027 20 EB       JR NZ,LOOP
30029 C9          RET
```

* *N del T.:* En la instrucción siguiente al byte 30006 (LD E, xx) deberás poner el valor correspondiente al color deseado —por ejemplo 06 te dará el amarillo (utiliza el número correspondiente a la tecla en donde está el color pero en Hex). LOOP = BUCLE.

Restas o substracciones de números y registros de otros registros

En Código Máquina, las instrucciones para restar números y registros de otros registros son exactamente las mismas que sus opuestas para la suma. Por esta razón la substracción es un concepto más sencillo de entender. Por supuesto que hay diferencias. Con la suma se pueden producir desbordamientos (overflow) pero con la resta se puede caer en la "escasez" (underflow). Dicho de otro modo, el número que "va" a ser restado es más pequeño que el número que se le va a "substraer". Prueba a restar 11 de 5 y tendrás una escasez de 6. Si ocurre una escasez en una resta entonces el señalizador de acarreo se marca con un uno pero, si no se produce esa escasez, entonces el señalizador de acarreo es puesta a cero. El siguiente factor que debemos recordar es que no hay un señalizador positivo y otro negativo. Cuando ocurre un desbordamiento en la suma, los números vuelven a comenzar de nuevo desde cero. Por ejemplo, si le sumas 5 a 255 entonces, en lugar de llegar a 256 y luego sumar 4, el ordenador se pondrá en cero y sumará 4. En la resta, volverá a 256 y luego restará la cantidad que sea.

La resta funciona del mismo modo que la suma pero teniendo en cuenta que la instrucción SUB A, B (SUB por SUBstracción) tomará el valor del registro B y se lo restará al registro A, y el resultado será almacenado en el registro A. El señalizador de acarreo será marcado (1) o puesto a cero (0) según las circunstancias.

Debido a que el comando SUB sólo se puede aplicar a registros sencillos, y solamente al acumulador, es normal (en lugar de escribir SUB A, B) que la instrucción "resta B a A" sea escrita como "SUB B". Esto puede parecer, al principio, un poco confuso pero te acostumbrarás a ello rápidamente. Como la resta no es el más sencillo de los temas al manejarlo con un formato poco familiar puede que, a estas alturas, sea una buena idea revisar el trabajo que hemos realizado anteriormente en el libro, simplemente para reforzar tu aprendizaje. Los diferentes comandos ensambladores y sus códigos equivalentes en lenguaje máquina se muestran a continuación:

MNEMONICO	HEX	DECIMAL
SUB (HL)	96	150
SUB A	97	151
SUB B	90	144
SUB C	91	145
SUB D	92	146
SUB E	93	147
SUB H	94	148
SUB L	95	149
SUB xx	D6xx	214, XX

Aquí abajo tenemos un rápido ejemplo de cómo se puede utilizar la operación de la resta:

```
ORG 30000
30000 3E 0B        LD A,11
30002 06 07        LD B,7
30004 90           SUB A,B
30005 4F           LD C,A
30006 06 00        LD B,0
30008 C9           RET
```

Es posible restar constantes numéricas al registro A. Por ejemplo, la instrucción SUB A, 100 restará 100 al número almacenado en el registro A. Debes fijarte que aunque hay instrucciones ADD para registros dobles no hay comandos de resta para registros dobles. A continuación ponemos un breve ejemplo de la utilización de la resta.

```
ORG 30000
30000 06 00          LD B,00
30002 3E 58          LD A,88
30004 D6 33          SUB 51
30006 4F             LD C,A
30007 C9             RET
```

La resta con acarreo

También, por otro lado, la resta con acarreo (SBC por **S**u**B**stracción con a**C**arreo) funcionará con registros dobles y, al igual que con ADD y ADC, sólo se puede alterar el valor del registro doble HL, o alterar el contenido del registro A. SBC A, C restará el valor del registro C al valor del registro A, y luego restará el valor de señalizador de acarreo a este resultado y lo almacenará en el registro A. Una vez más el comando de "resta con acarreo" puede ser usado al restar dos números uno del otro y pudiera producirse un desbordamiento.

Si volvemos a recordar el modo en que comenzamos a restar por primera vez unos números de otros, podemos ver cómo funciona esto. Si tuviéramos el número 21 y deseáramos restarle 19, procederíamos de esta forma. Primero restaríamos 9 al 1. Esto no es así exactamente —si el número sigue siendo positivo— o sea que la solución es "pedir" 1 (de valor 10) a la columna de la izquierda. Ahora "restamos 9 al 11", lo que nos da un resultado de "2". Hay que tener en cuenta que la columna siguiente se ha reducido en 1, ocurriendo lo que se denomina "una escasez" ("underflow"). Por tanto, es necesario restar "1 al 2", siendo 1 el resultado —pero, debido a que tenemos que restar el señalizador de acarreo a este resultado, la respuesta es 0 y el resultado final es 2.

Serás capaz de deducir del listado inferior, que al utilizar la

"resta con acarreo" en un registro doble se requieren dos bytes, y una operación simple con un registro sencillo requiere un byte —lo cual hace que ambos comandos sean muy compactos. Los códigos del lenguaje ensamblador y los del lenguaje máquina se muestran a continuación:

MNEMONICO	HEX	DECIMAL
SBC A, (HL)	9E	158
SBC A, A	9F	159
SBC A, B	98	152
SBC A, C	99	153
SBC A, D	9A	154
SBC A, E	9B	155
SBC A, H	9C	156
SBC A, L	9D	157
SBC A, xx	DExx	222, XX
SBC HL, BC	ED42	237, 66
SBC HL, DE	ED52	237, 82
SBC HL, HL	ED62	237, 98

Operaciones con el señalizador de acarreo

A menudo es útil, cuando empleamos instrucciones tales como las de resta con acarreo y las de suma con acarreo, que sea posible determinar el contenido del señalizador de acarreo. Desgraciadamente no es posible poner a cero, es decir, poner directamente a cero el señalizador de acarreo pero se puede hacer con un sencillo truco de Código Máquina. Ya sabemos que cuando se suma un número al acumulador, el señalizador de acarreo se fija según haya o no haya un desbordamiento. Si sumamos 0 al acumulador entonces no es posible que pueda haber desbordamiento. POR CONSIGUIENTE HACIENDO EXACTAMENTE ESTO PODEMOS PONER A CERO EL SEÑALIZADOR DE ACARREO. Esto sucede porque el señalizador de acarreo siempre se pone a cero cuando no hay desbordamiento. Probablemente éste sea el comando más útil para aprovecharnos del señalizador de acarreo. El utilizar los comandos antes mencionados puede cambiar enormemente el resultado en el caso

de que el señalizador de acarreo active un 1 cuando debería estar puesto a cero. Poner en 1 al señalizador de acarreo es una tarea mucho más fácil. Hay una instrucción que hace esto. Es la instrucción SCF que significa "Set Carry Flag" ("Marcar el Señalizador de Acarreo"). El código en lenguaje máquina de este comando es 37 hexadecimal.

Con esto concluye el capítulo. Hemos visto dos formas importantes de manipular los registros y maneras con las que tomar números de las posiciones de memoria y colocarlos en los registros. También hemos visto comandos más complicados —tales como ADC (Suma con Acarreo)— y la forma de solucionar problemas activando y poniendo a cero el señalizador de acarreo. Espero que a estas alturas ya seas capaz de escribir un pequeño programa en Código Máquina utilizando estos comandos.

¡¡Lo divertido del Código Máquina es entrar de lleno en él!!

Capítulo 6

PROGRAMAS ENSAMBLADORES, DESENSAMBLADORES Y CORRECTORES (DEPURADORES)

En este capítulo nos vamos a tomar un pequeño descanso del aprendizaje del Código Máquina. En su lugar echaremos un vistazo a los tres tipos principales de programas que se han escrito específicamente para ayudar al programador de Código Máquina. Ahora ya hemos avanzado bastante en el aprendizaje sobre cómo escribir pequeños programas en Código Máquina, pronto te preguntarás si estás preparado para emprender programas más arriesgados. Esto te conducirá a escribir rutinas en Código Máquina más largas y, por esta razón, será útil tener alguna ayuda especial para hacerte la cosa más fácil. La mayoría de estas ayudas, o facilidades, también están escritas en Código Máquina. Por consiguiente queda claro que el programador que las escribió probablemente sea alguien con grandes conocimientos sobre este tema, y las instrucciones de las operaciones no tienen por qué estar necesariamente dadas con la sencillez del lenguaje habitual. En consecuencia, cuando tenga que descifrar los manuales de instrucciones del ensamblador corrector o del desensamblador, el principiante podría encontrarse en clara desventaja. Por esta razón, al final de este capítulo, he incluído una breve sección explicando cuidadosa y exactamente cómo utilizar ejemplos concretos de cada uno de estos tipos de programas.

La elección de estos programas depende en gran medida de las preferencias personales y de la experiencia. Para tu información, todos los programas de este libro fueron ensamblados/desensamblados con la ayuda de los programas de ACS Software (Gran Bretaña).

El lenguaje máquina y el lenguaje ensamblador, comparados

En este momento es importante que sepamos exactamente qué queremos decir con el lenguaje máquina y con el lenguaje ensamblador, y que son, por tanto, adaptables a todo lo que hemos estado hablando hasta ahora. Por eso vamos a hacer una rápida recapitulación. (Si estás convencido de que sabes exactamente lo que significa todo esto y las diferencias entre el lenguaje máquina y el lenguaje ensamblador, entonces puedes saltarte esta sección).

El "lenguaje máquina" está hecho para números y nada más que números. El mismo ordenador entiende estos números y puede ejecutarlos como comandos. Desgraciadamente el hecho de tener que recordar una gran cantidad de números, y luego ser capaces de saber exactamente qué es lo que hacen, no es una tarea demasiado fácil para los humanos. Por esta razón se ha creado el "lenguaje ensamblador". Cada comando en lenguaje ensamblador se corresponde directamente con un comando en lenguaje máquina, excepto que en lenguaje ensamblador estas instrucciones están escritas en mnemónicos, los cuales son una abreviatura de dichos comandos. Por ejemplo: si queremos ejecutar el comando "Cargar el registro A con el contenido del registro B", esto se puede realizar en lenguaje máquina introduciendo simplemente el código 78, y ejecutándolo. Pero el hecho de recordar el número 78 y que se corresponde con la instrucción anterior no es una tarea fácil. Una forma de evitar esto es tener una lista de todas las instrucciones y todos los códigos para que puedas ver cada comando concreto y su correspondiente código en lenguaje máquina.

El primer problema que se plantea es que si tuviéramos que completar cada comando con ciertas explicaciones esto nos ocuparía demasiado sitio. Por eso, en lugar de hacer todo ese trabajo, tenemos los MNEMONICOS. Un mnemónico es una abreviatura de su

comando. En lugar de tener "Cargar el registro A con el contenido del registro B" tenemos la abreviatura LD A, B. Ya te habrás dado cuenta de que es mucho más fácil acordarse de esta abreviatura que del código en lenguaje máquina, y de que ocupa mucho menos espacio que una explicación completa. Ahora, si queremos utilizar este método de referencia, puede hacerse bastante fácilmente listando el mnemónico al lado del código en lenguaje máquina. (Esto es lo que se ha hecho al final del libro). Si nos aprendiéramos todos los mnemónicos de las instrucciones en lenguaje máquina tendríamos lo que se llama un lenguaje ensamblador. Con otras palabras, un lenguaje ensamblador se hace a base de los mnemónicos (o, más correctamente, mnemotécnicos) del lenguaje máquina. Por esta razón podemos observar que el lenguaje ensamblador no es una adaptación de un lenguaje máquina al igual que el BASIC, ya que sólo hay un único mnemónico para cada instrucción en lenguaje máquina y viceversa.

El trabajo que realiza un ensamblador

Por consiguiente, lo que tenemos es un lenguaje que podemos entender con bastante facilidad y que podemos utilizar con ayuda de listados convenientemente formulados. De cualquier manera, aunque ya dispongamos de un lenguaje más adecuado, todavía requerirá que le dediquemos algo de tiempo y esfuerzo a la "transformación". ¿Por qué no hacemos que nuestro inteligente ordenador haga por nosotros todo el trabajo pesado de transformar los mnemónicos en códigos de lenguaje máquina? Esto es precisamente lo que un ENSAMBLADOR es capaz de hacer por nosotros: es un programa que ensambla (transforma) los mnemónicos en códigos de lenguaje máquina.

Otra forma de imaginarnos este trabajo es viendo el modo en que trabaja un establecimiento de comidas rápidas. Cuando entras pides, según la carta, y en esa carta hay varios platos, por ejemplo, hamburguesas, o lo que sea. Una vez que pides lo que quieres al camarero, él o ella no anota tu pedido sino que, por el contrario anota el número correspondiente a un producto previamente definido. Por tanto, lo que están haciendo es "ensamblar" tu pedido

de forma que los que están en la cocina lo entenderán y lo utilizarán más rápidamente.

Todos los ensambladores comunes al Spectrum funcionan basados en este principio: primero introduces tu programa en lenguaje ensamblador y luego, una vez finalizado, ordenas al programa que lo ensamble en lenguaje máquina. Luego, los códigos en lenguaje máquina, son colocados en una zona concreta de la memoria según lo desees. Por supuesto también depende de la memoria disponible del ordenador. Un programa ensamblador es un tipo de programa bastante complicado y algunos de los ensambladores del mercado sólo son compatibles con el Spectrum de 48K. Esto me ha influido en la elección de un Ensamblador para escribir este libro, ya que los programas ACS funcionan indistintamente en las versiones de 16K y 48K.

Si ojeas en las revistas sobre los Sinclair o de ordenadores personales, te habrás fijado en varios anuncios de ensambladores. El precio de un ensamblador normalmente cuesta un par de libras (unas 500 Pts) más que un programa de juegos. Esto se debe a dos razones. En primer lugar son, la mayoría de ellos, más complicados que los juegos; y, en segundo lugar, tampoco son productos para un "mercado de masas" y de los que no se puede esperar que sean vendidos en grandes cantidades. Créeme, vale la pena comprarlos. Son increiblemente útiles para un auténtico programador de Código Máquina.

La misión de los desensambladores

Un desensamblador hace exactamente lo contrario de un ensamblador: transforma el Código Máquina en códigos de lenguaje ensamblador.

Principalmente existen dos formas en las que se puede utilizar un desensamblador:

1. Como una ayuda para comprobar que lo que se ha puesto en la memoria del ordenador coincide con lo que tú querías poner.
2. Para ver qué es lo que han escrito otros en Código Máquina, y cómo han llegado a resolver ciertos problemas.

Por ejemplo, puedes querer desensamblar ciertas zonas de la ROM del Sinclair con el fin de entender mejor qué es lo que allí ocurre y, si es posible, utilizar dichas rutinas dentro de tus programas. También, en muchas revistas, cuando se lista un programa en Código Máquina éste aparecerá sólo en código Hex o en Código Máquina. Te recomiendo que más adelante cargues algún programa de otra persona y luego lo desensambles. Así tendrás una idea de cómo otros escriben los programas. Hay mucho que aprender viendo cómo los demás utilizan las distintas instrucciones.

Con el ensamblador y el desensamblador utilizados para este libro, es posible, si tienes un ordenador de 48K, cargar ambos al mismo tiempo. De nuevo, a la hora de elegir tus accesorios, recuerda la utilidad de este tipo de "compatibilidad". No es posible introducir al mismo tiempo un ensamblador y un desensamblador en un ordenador de 16K debido sencillamente a que no habría memoria suficiente. De todas formas y en tales circunstancias es posible cargarlos uno cada vez. Ya que es posible cargar (load) y grabar (save) nuestros propios programas en Código Máquina, independientemente de los programas que ya se encuentren en el ordenador, podemos ensamblarlos utilizando el ensamblador, grabarlos, y luego volver a cargarlos con el desensamblador para desensamblarlos (¡menudo trabalenguas!).

Los programas correctores (depuradores)

El tercer y último tipo de programa del que vamos a tratar no es uno que podamos usar cuando inicialmente escribimos el programa en Código Máquina sino que es útil para corregir y para alterar el programa una vez que ha sido ensamblado. Cuando compras un programa "corrector" ("debugging program") normalmente no estás comprando un programa común. Un buen programa corrector de Código Máquina debe contener alguna de estas opciones:

1. Una opción que te permita ejecutar la rutina en Código Máquina, instrucción a instrucción, y que sea capaz de mostrar el contenido de todos los registros después de cada paso. Se trata de una posibilidad extremadamente útil cuando nos es necesario localizar un

error ("bug") en un programa, o cuando deseas ver cómo funcionan exactamente algunas rutinas.

2. Que sea capaz de introducir un "punto de ruptura" en cualquier lugar del programa, es decir, que ejecute el programa hasta ese punto y luego se detenga. Este comando puede ser utilizado de varios modos. Puede que no desees saber el contenido de los registros después de cada paso de programa, especialmente si se trata de uno muy largo sino que deseas conocer el estado de todos los registros y de los señalizadores después de cierta operación, o una vez que se hayan ejecutado un conjunto de operaciones. Para continuar ejecutando el programa debería hacerse pulsando una tecla determinada.

3. Que sea capaz de cambiar el contenido de un registro concreto mientras el programa esté detenido en un punto de ruptura, lo cual puede conseguirse bastante fácilmente. Esto puede ser muy útil especialmente en dos casos concretos. En primer lugar, si desearas saber cuál sería el resultado si hubieras cargado unos registros con ciertos valores. En segundo lugar, podría ser útil la posibilidad de simular ciertas condiciones cuando se introducen valores que son demasiado grandes para ser manejados por dicho programa pero deseas saber si las rutinas que los van a manejar van a seguir funcionando o no.

4. Que tenga capacidad para ejecutar tu programa desde el programa monitor (es un programa corrector y ensamblador). Esto puede parecer obvio pero mientras que estás utilizando el monitor no siempre resulta fácil volver al BASIC y utilizar el comando USR.

5. La facultad de transformar números de hexadecimal a decimal, dentro del mismo programa corrector.

6. Tres útiles comandos que, cuando los empleemos al mismo tiempo, permitan al programa corrector alterar el programa en Código Máquina ya ensamblado. Esto nos permite la inspección del contenido de cualquier dirección y modificarla si es preciso. De forma adicional, en muchos ensambladores, si mantienes el dedo sobre la tecla ENTER, el contenido de las sucesivas direcciones aparece automáticamente en la pantalla, pero por supuesto no se puede intro-

ducir un nuevo valor a no ser que sea introducido antes de pulsar dicha tecla. Si resulta que deseas introducir un comando dentro de un programa en Código Máquina, suele ser bastante difícil una vez se ha ensamblado ya que, por supuesto, sería necesario subir o bajar la mitad del programa. En muchos programas correctores resulta fácil introducir una nueva instrucción ya que basta con decirle al ordenador dónde la deseas introducir exactamente, y luego cuántos bytes deseas introducir en el espacio recién creado y, finalmente, el código o códigos a introducir. La instrucción complementaria a la anterior es, por supuesto, «borrar», que puede encontrarse en la mayoría de los programas correctores. Trabaja exactamente al revés que el comando de «introducción».

Lo que hemos visto no es más que una breve guía del contenido de un programa corrector aunque normalmente encontrarás comandos adicionales distintos de los que acabamos de ver. La forma en que opera un programa corrector varía de acuerdo con el programa que utilices.

A menudo un ensamblador o un desensamblador pueden ser fundidos con un programa corrector, o viceversa. Por ejemplo, es habitual encontrar en un programa corrector que tiene una función de desensamblaje. Otro ejemplo de esta costumbre es el ensamblador ofrecido por Picturesque. Al mismo tiempo que puedes ensamblar códigos, también puedes editarlos a través de varios comandos muy complejos. El programa monitor, que básicamente es un programa ensamblador y corrector, puede utilizarse, al mismo tiempo, como un programa ensamblador/editor, consiguiendo así una combinación muy potente. Para una mayor sencillez, y para conseguir que todo fuera más barato, los programas ensamblador/desensamblador/corrector utilizados en este libro eran programas separados, y se hallaban disponibles en cintas cassette diferentes.

Un programa corrector no tiene por qué estar escrito necesariamente en Código Máquina. Puede ser que decidas no comprarte ningún programa de este tipo por ahora. La mejor solución es utilizar el programa monitor listado en las páginas siguientes. Esto te permitirá introducir el Código Máquina en el ordenador y editarlo satisfactoriamente, a pesar de que tendrás que introducirlo con sus códigos en lenguaje máquina. Se puede conseguir la conversión de

los mnemónicos del lenguaje ensamblador utilizando la sección de referencia del final del libro.

Una vez que hayas tecleado este programa recuerda guardarlo en una cinta cassette (SAVE) antes de utilizarlo, y una vez estés seguro de que funciona correctamente, haz una copia separada en otra cinta, por si alguna de las cintas resulta dañada por cualquier motivo.

```
   5 CLEAR 31999
  10 PRINT "***MONITOR PARA EL S
PECTRUM***"'"        POR JAMES WAL
SH '83"
  20 PRINT AT 3,4; FLASH 1;"LAS
LETRAS EN MAYUSCULAS"
  30 REM CARGADOR DE C/M
 100 INPUT "DIRECCION INICIAL (D
EC)? ";DIR
 110 PRINT AT 5,0;
 120 PRINT DIR;"...";
 130 LET N=PEEK DIR: GO SUB 1400
 140 INPUT Z$
 145 PRINT "....";Z$
 150 IF Z$="fin" OR Z$="FIN" THE
N  GO TO 800
 160 IF Z$="p" OR Z$="P" THEN  C
OPY : CONTINUE : GO TO 120
 170 IF Z$="s" OR Z$="S" THEN  G
O TO 280
 190 IF Z$="v" OR Z$="V" THEN  G
O SUB 1600: GO TO 120
 200 IF Z$="j" OR Z$="J" THEN  L
ET DIR=DIR-1: GO TO 120
 210 IF Z$="n" OR Z$="N" THEN  G
O TO 100
 220 IF Z$="z" OR Z$="Z" THEN  L
ET Z$="00"
 230 LET TOT=(CODE Z$(1)-48)*16
 240 IF TOT>9*16 THEN  LET TOT=T
OT-7*16
 250 LET B=CODE Z$(2)-48: IF B>9
 THEN  LET B=B-7
```

```
 260 LET TOT=TOT+B
 270 POKE DIR,TOT
 280 LET DIR=DIR+1
 290 GO TO 120
 800 STOP
1400 REM SUBRUTINA HEX/DEC
1410 LET X=INT (N/16)
1420 LET Y=((N/16)-INT (N/16))*1
6
1430 IF X>9 THEN  LET X=X+7
1440 IF Y>9 THEN  LET Y=Y+7
1450 LET X=X+48: LET Y=Y+48
1460 PRINT CHR$ X;CHR$ Y;
1470 RETURN
1600 REM SUBRUTINA HEX/DEC
1605 INPUT "NUMERO A CONVERTIR?"
;N
1607 LET A=N
1610 LET X=INT (N/16)
1620 LET Y=((N/16)-INT (N/16))*1
6
1630 IF X>9 THEN  LET X=X+7
1640 IF Y>9 THEN  LET Y=Y+7
1650 LET X=X+48: LET Y=Y+48
1660 PRINT A;" = ";CHR$ X;CHR$ Y
1670 RETURN
```

Cómo funciona un programa monitor

Vamos a ver cómo funciona el programa monitor cuya lista acaba de incluirse. Es más corto que los programas comerciales que hemos analizado hasta ahora, y seguramente es el más fácil de manejar. La primera cosa que ha de señalarse es que la línea 5 es utilizada para fijar el RAMTOP (tope de la RAM) a través de la instrucción CLEAR. Por lo tanto, es necesario ajustar este valor, según la zona de memoria utilizada por la rutina de Código Máquina. Ahora ejecuta el programa. Se te pedirá que introduzcas la posición inicial. Hazlo. Esta dirección inicial será presentada en la pantalla, seguida por su contenido en Hex. Ahora ya se puede introducir un nuevo

valor (en Hex), acordándote de utilizar solamente las MAYÚSCULAS. Además puedes utilizar uno de los siguientes comandos:

P — Hace una copia de la pantalla por la impresora.

S — Deja como está el contenido de esa posición y salta a la siguiente.

V — Convierte un número decimal en uno hexadecimal.

J — Salta hacia atrás una posición.

N — Comienza desde una nueva posición.

Z — Pone a cero el valor de la posición en la que se encuentra.

Una vez se ha introducido el valor, o se han ejecutado los comandos P, V o Z, se visualizará la nueva posición y su contenido. Para terminar este proceso, hay que teclear «fin» cuando se pide un valor. Piénsese que este programa es corto y relativamente sencillo y resulta verdaderamente valioso cuando se carece de un Ensamblador o de un programa de depuración comercial.

Cómo utilizar un ensamblador

El ensamblador puede cargarse en el ordenador en la forma habitual. Si tienes ambas versiones de 16K y 48K asegúrate de que cargas la copia adecuada en el ordenador. Es importante que no cargues la copia equivocada. El ensamblador que se ha utilizado en la confección de este libro se llama Ultra Violet.

Si decides utilizar el mismo ensamblador probablemente querrás beneficiarte de algunas explicaciones adicionales sobre su funcionamiento.

Para cargar el Ultra Violet ejecuta LOAD»». (La versión para el Spectrum de 16K se carga en tres partes.) Para comenzar a utilizar el ensamblador pulsa cualquier tecla del teclado. Esto despejará la memoria (CLEAR) del ordenador «por debajo» de la zona donde está almacenado el ensamblador. El ensamblador está almacenado en la «parte alta» de la memoria, a salvo de los programas en BASIC. Allí es donde permanecerá y donde podrá ser llamada desde el BASIC. Una vez hecho esto, estarás en disposición de utilizar los comandos del BASIC. Ahora ya se puede introducir el Código Máquina en un programa casi como si fuera BASIC. La única diferencia es que tendremos que poner una sentencia REM antes del propio Código Máquina. La línea que va a contener los comandos de Código Máquina

tendría este aspecto: Primero el número de línea, luego la sentencia REM y luego los mnemonicos del lenguaje ensamblador.

Lo primero que tenemos que hacer cuando utilizamos este ensamblador es decidir dónde deseamos colocar exactamente el Código Máquina una vez que éste haya sido ensamblado. Hay dos alternativas: primero, y probablemente sea lo más útil, es colocarlo en algún lugar de la memoria por encima del programa BASIC pero por debajo del ensamblador.

> *NOTA:* El mismo ensamblador utiliza, en la versión de 48K, las direcciones desde la 60000 hacia arriba, y en la versión de 16K desde la 27500 hacia arriba. De aquí que no sea posible ensamblar códigos directamente en estas posiciones, aunque es posible resolver este problema como veremos más adelante.

La segunda alternativa es la de ensamblar el Código Máquina en una sentencia REM al principio del programa. El hecho de que vaya al principio es sólo por comodidad. La dirección habitual del primer carácter, después de la sentencia REM colocada en la primera línea, es 23760. Si hubiéramos decidido colocar el Código Máquina en una sentencia REM, lo cual por ahora no es aconsejable y resulta bastante complicado en el Spectrum, entonces deberíamos dejar «espacios» o «caracteres» suficientes después de la primera sentencia REM, para contener al Código Máquina. Debes recordar que si deseas guardar una rutina de Código Máquina de 25 bytes en una sentencia REM, esta sentencia REM debe contener, por lo menos, 25 caracteres inmediatamente después de la propia sentencia REM.

Este ensamblador puede reconocer todos los mnemónicos estándar del Código Máquina del Z80 que están en minúsculas, exactamente tal y como están en la parte final del manual de programación de BASIC del ZX Spectrum que se entrega con el ordenador. Sólo hay un par de excepciones a esta regla, que no tienen demasiada importancia y que veremos más adelante. Una cosa importante en la que nos debemos fijar es que todos los números deberemos introducirlos en decimal, aunque una vez que sean ensamblados, serán listados en hexadecimal.

Cómo pasar tu programa al ensamblador

La primera instrucción que se ensambla en cualquier programa

de Código Máquina es el comando GO. No se trata de un mnemónico del Código Máquina del Z80. Sencillamente sirve para decirle al ensamblador dónde vamos a poner el programa que vamos a ensamblar. La línea siguiente debe contener la dirección en la que ha de ser ensamblado el Código Máquina. Esto se hace a través del comando ORG seguido de la dirección. ORG tampoco es un mnemónico. Sólo dice al programa dónde colocar los códigos ensamblados. ORG viene de «ORiGen». Si omitiéramos alguno de los dos comandos anteriores se originaría un código o mensaje de error cuando tratáramos de ensamblarlo. Ya estás preparado para introducir los mnemónicos de tu programa en Código Máquina. Puedes poner más de un mnemónico en cada línea pero separados por punto y coma (;), aunque probablemente sea más fácil y quede más claro si sólo pones una instrucción por línea. Ya habremos dejado espacio al principio del programa para el Código Máquina (pero sólo en el caso de que queramos ponerlo en una sentencia REM). La instrucción GO le dice al ordenador dónde van a ser almacenados los códigos a ensamblar, y también ya le hemos dicho, a través del comando ORG, en qué sitio de la memoria queremos poner los códigos ensamblados. Ahora ya estamos preparados para introducir el programa en lenguaje ensamblador. Muchas veces, durante un programa, puede que desees poner algún tipo de comentario o explicación de lo que estás haciendo. Esto puede conseguirse cuando utilizas un ensamblador colocando un signo de admiración (!) después del REM y luego escribiendo el comentario. El signo de admiración se utiliza para comunicar al ordenador que los códigos de la palabra posterior no son para ensamblar sino que están allí con el fin de ser utilizados por el programador. Si accidentalmente omites el signo de admiración, el ensamblador tratará de transformar en Código Máquina lo que hayas escrito, y probablemente detendrá la marcha del ensamblador y mostrará un código de error.

Ahora ya puedes introducir el programa para que sea ensamblado. Esto habrá de hacerse con letras minúsculas, las cuales ya estarán presentes cuando conectemos el ordenador, para que el ordenador pueda entender qué es lo que estás haciendo.

NOTA: En las sentencias con COMENTARIOS puedes utilizar tanto las mayúsculas como las minúsculas, aunque tengas que

usar obligatoriamente las minúsculas para que las instrucciones sean ensambladas.

Una vez que hayas introducido todos los mnemónicos del lenguaje ensamblador para que sean ensamblados debes poner la palabra «Finish» después de la sentencia REM. Esto le comunica al ordenador que has llegado al final de la rutina que estabas ensamblando. «Finish» tampoco es un mnemónico del Z80.

Ahora ya deberías tener la totalidad del programa en Código Máquina en forma de lenguaje ensamblador, el cual ya está preparado para el ensamblaje en lenguaje máquina. Antes de hacer esto será útil comprobar que no hemos cometido ningún error al introducir el programa. En caso de que los haya, obviamente, será cuestión de editar la línea correspondiente y cambiar la parte errónea tal y como lo harías en BASIC. Para ordenar al programa ensamblador que deseas ensamblar los mnemónicos ya introducidos, escribe RANDOMIZE SPACE USR 60000, si tienes un Spectrum de 48K, o RANDOMIZE SPACE USR 27500 si es que tienes un Spectrum de 16K. Ahora los mnemónicos aparecerán en la pantalla junto a sus códigos en lenguaje máquina. Si hubiera algún error, es decir sentencias que no pueda entender el ensamblador, entonces aparecerá un código de error y se te pedirá que rectifiques dicho error. Por favor, fíjate que un ensamblador no comprobará el programa por ti. Otra cosa importante es que cuando se está ensamblando el programa, el listado del programa en Código Máquina se escribe dos veces. Esto es así debido a que se hace un «doble paso» por el ensamblador. Esto significa que revisa el programa dos veces antes de poner la versión final en la memoria. Las razones para hacer esto se pondrán de manifiesto más adelante.

Ahora ya sabemos, en teoría, lo que podemos hacer. Vamos a ver cómo lo ponemos en práctica. Lo primero que tenemos que recordar es que queremos poner el lenguaje máquina, no en una sentencia REM al principio del programa, sino, digamos, en la posición 30000. El programa que vamos a ensamblar se muestra a continuación y es una rutina que hace el «scroll», pixel por pixel, hacia la derecha.

```
LD HL,22527
PUSH HL
LD DE,23328
LD BC,32
```

```
                    LDDR
                    POP HL
HL=DIRECCION MAS BAJA DEL BLOQUE
   INFERIOR
B=NUMERO DE BLOQUES
                    LD B,3
A
                    PUSH BC
                    LD B,8
B
                    PUSH BC
                    PUSH HL
                    LD B,8
C
                    PUSH BC
                    PUSH HL

                    PUSH HL

                    POP DE
                    DEC H
                    LD BC,32
                    LDDR
                    POP HL
                    DEC H
                    POP BC
                    DJNZ,C
                    INC H
                    LD DE,1760
                    PUSH HL
                    ADD HL,DE
                    POP DE
                    LD BC,32
                    LDDR
                    POP HL
                    LD DE,32
                    SBC HL,DE
                    POP BC

                    DJNZ,B
                    LD DE,1792
```

```
                    SBC HL,DE
                    PUSH HL
                    LD DE,32
                    ADD HL,DE
                    PUSH HL
                    POP DE
                    POP HL
                    PUSH HL
                    LD BC,32
                    LDDR
                    POP HL
                    POP BC
                    DJNZ,A
                    LD DE,32
                    ADD HL,DE
                    EX DE,HL
                    LD HL,23328
                    LD B,32
D
                    LD A,(HL)
                    NOP
LD A,0 PARA EVITAR BUCLE SIN FIN
                    LD (DE),A
                    DEC DE
                    DEC HL
                    DJNZ,D
                    RET
```

El programa anterior aparece en lenguaje ensamblador. El trabajo del ensamblador consiste en convertirlo en códigos de lenguaje máquina y su primera operación es colocar todo esto en el ensamblador. Lo primero que hemos de hacer es cargar en ensamblador. Esto se consigue escribiendo LOAD »» asegurándonos de que cargamos el lado correcto (48K ó 16K) de acuerdo con el tamaño del ordenador. Según el tamaño del ordenador cargar el ensamblador le llevará más o menos tiempo pero, en cualquier caso, menos de un minuto. Si estás utilizando el ensamblador Ultra Violet, entonces aparecerá en la pantalla una gran «grieta» pero desaparecerá tan pronto como pulses una tecla, dejando el habitual mensaje de encendido, es decir (c) 1982 Sinclair Research Ltd. Ahora ya estás pre-

parado para escribir las instrucciones en lenguaje ensamblador a través de un programa en BASIC. Por el momento, puedes abordar esta operación como si se tratara de un programa BASIC. Es muy importante recordar que todas las instrucciones en lenguaje ensamblador, y las instrucciones que se utilizan para decirle al mismo ensamblador lo que tiene que hacer, han de ser introducidas en una línea pero después de una sentencia REM. Esto se debe a que el ordenador por sí mismo no puede ejecutar dichas instrucciones ya que sólo son reconocibles por el ensamblador y no por el BASIC.

Ahora debemos comunicar al ensamblador que ya hay un programa en lenguaje ensamblador preparado para ser ensamblado. Esto se hace introduciendo la instrucción «GO» como primera línea del programa. Por ejemplo: 5 REM go. Lo que debemos especificar a continuación es en qué lugar de la memoria queremos poner la rutina ensamblada en Código Máquina. Esto se hace a través de la instrucción «ORG», seguida de la dirección en la que queremos poner el primer byte del primer código ensamblado; el resto de los códigos se colocarán secuencialmente después de esta dirección. Así, por ejemplo, si nuestra rutina tiene una longitud de 10 bytes, y le comunicamos al ordenador que empiece a ensamblar en la posición 28000, entonces la colocará en las direcciones 28000 hasta la 28009 (en total diez bytes). Debido a que queremos colocar el anterior programa ensamblado en las posiciones 30000 en adelante, la siguiente línea del programa BASIC que introduciremos será: 8 REM org 30000.

Ahora ya estamos casi preparados para escribir las instrucciones en lenguaje ensamblador pero, antes de hacerlo, nos será muy útil poner un comentario al principio del programa de forma que, si volvemos a verlo más tarde, podamos reconocerlo. La forma de poner un comentario en el programa es escribiendo un signo de admiración (!) después del número de línea y de la sentencia REM, y luego poner dicho comentario. Para el programa anterior bastaría decir que se trata de un programa que hace el «scroll» descendente por pixels y lo haríamos así: 12 REM ! Scroll descendente por pixel.

Ya lo tenemos todo preparado y dispuesto para introducir las instrucciones en lenguaje ensamblador. Acuérdate, tal y como harías en un programa en BASIC, de poner los números de línea para cada rutina, y no te olvides de poner la sentencia REM antes de

cada instrucción. El último comando que debemos dar al ensamblador es el que le comunica que hemos terminado con los códigos a ensamblar. Esto se realiza utilizando el comando «Finish». Puede ponerse en una línea tal y como hicimos más arriba: número de línea REM Finish. Una vez introducido en el ordenador, comienza el ensamblaje.

Si por alguna razón has tenido algún problema al escribir el programa y deseas comprobar lo que has escrito, a continuación se muestra el programa BASIC tal y como debe aparecer. No te preocupes si los números de línea son exactamente los mismos que en el mío, solamente asegúrate de que todas las instrucciones están en el mismo orden.

```
   1 REM GO
   2 REM ORG 30000
  10 REM LD HL,22527
  20 REM PUSH HL
  30 REM LD DE,23328
  40 REM LD BC,32;LDDR;POP HL;!H
L=DIRECCION MAS BAJA DEL BLOQUE
INFERIOR
  50 REM !B=NUMERO DE BLOQUES;LD
 B,3;A;PUSH BC;LD B,8;B;PUSH BC;
PUSH HL;LD B,8;C;PUSH BC;PUSH HL
;PUSH HL;POP DE;DEC H;LD BC,32;L
DDR;POP HL;DEC H;POP BC;DJNZ,C;I
NC H;LD DE,1760;PUSH HL;ADD HL,D
E;POP DE;LD BC,32;LDDR;POP HL;LD
 DE,32;SBC HL,DE;POP BC;DJNZ,B;L
D DE,1792;SBC HL,DE;PUSH HL;LD D
E,32;ADD HL,DE;PUSH HL;POP DE;PO
P HL;PUSH HL;LD BC,32;LDDR;POP H
L;POP BC;DJNZ,A
  60 REM  LD DE,32;ADD HL,DE;EX
DE,HL;LD HL,23328;LD B,32;D;LD A
,(HL);NOP;'LD A,0 PARA EVITAR BU
CLE SIN FIN;LD (DE),A;DEC DE;DEC
 HL;DJNZ,D;RET
1000 REM FIN
```

El ensamblaje se consigue accediendo a la rutina en Código Máquina que hemos cargado previamente en el ordenador. Al igual que hay dos versiones de este programa, también hay dos direcciones diferentes en las que se almacena la rutina ensambladora. Si tienes un Spectrum de 16K, debes escribir el comando siguiente:

RANDOMIZE USR 27500

Si tienes un Spectrum de 48K, debes ejecutar el comando:

RANDOMIZE USR 60000

Importante: Por favor, recuerda que las instrucciones de ensamblaje que hemos visto, y también la forma en que introducimos nuestra rutina en lenguaje ensamblador dentro del programa ensamblador, sólo son válidas si estás utilizando el ensamblador Ultra Violet. Hoy por hoy existen varios ensambladores en el mercado, y es probable que aparezcan todavía más desde que este libro se ponga a la venta y me es imposible describir la manera en que funcionan todos ellos, teniendo en cuenta que, a menudo, existen grandes diferencias a la hora de operar. He preferido concentrarme en el manejo de un ensamblador en particular y creo que éste es el más fácil de utilizar. Si ya tienes un ensamblador, o decides comprar uno distinto al ACS que he venido utilizando, entonces es conveniente que te estudies cuidadosamente su manual de instrucciones. Esto también significa que el listado mostrado anteriormente probablemente no valga para ser utilizado por otro ensamblador. Por supuesto que la rutina que ha sido listada un poco antes se ejecutará de igual forma en aquellos ensambladores que utilices.

Volviendo a la tarea que nos ocupa. Ya tenemos preparado el programa en BASIC, y ya sabemos cómo ordenar al programa ensamblador que ensamble las instrucciones que hemos escrito. Por lo tanto, ¿por qué no ver cómo funciona? Una vez que hayas ejecutado el comando de ensamblaje, mostrado recientemente, se te ofrecerá una imagen multicolor en la pantalla (que será «multigris» si es que utilizas una televisión en blanco y negro). Pero el ensamblaje todavía no ha finalizado. Si pulsas la tecla «P», entonces aparecerá en la impresora un listado de la rutina en Código Máquina. Si pulsas la tecla espaciadora (SPACE) se detendrá el ensamblaje y

pulsando una tecla cualquiera éste continuará. Es esencial recordar que sólo puedes utilizar el comando «P» para imprimir con la impresora una vez que haya sido presentada, por segunda vez, la rutina en Código Máquina. Una vez pulsada la tecla ENTER, y no habiendo aparecido en la parte baja de la pantalla ningún mensaje de error, ya estarás preparado para ejecutar la rutina en Código Máquina de la dirección 30000 en adelante de la forma habitual (RANDOMIZE USR 30000 o PRINT USR 30000).

Si se produce algún error en el programa ocurrirá lo siguiente:

— Si faltan alguno de los comandos «Go», «Finish» u «Org», el error aparecerá antes de comenzar el ensamblaje.

— Si se detecta un error durante el ensamblaje, entonces el ensamblador se parará con uno de los dos mensajes de error siguientes.

El primero es un mensaje de error parpadeante en el que se señala el número de líneas, número de sentencia y tipo de instrucción en la que ha sido detectado el error. Es el momento de volver a revisar el programa BASIC con el objeto de encontrar el error que se ha producido.

El segundo mensaje de error es uno de los mensajes Sinclair. Hay tres posibilidades. Si introduces un número equivocado, por ejemplo, si tratas de almacenar un número mayor que 256 en un registro sencillo o mayor que 65535 en un registro doble, el ensamblador le restará repetidamente 256 ó 65536, hasta que alcanza un número utilizable. Desafortunadamente esto no sucederá si has establecido un desplazamiento demasiado grande dentro de un salto relativo. A pesar de que todavía no hemos tratado acerca de los saltos relativos, lo cual haremos en breve, bueno será que lo tengamos en cuenta para más adelante. En caso de que el ensamblador no consiga alcanzar un resultado válido, como, por ejemplo, en el caso del salto relativo, entonces aparecerá el error «B Integer out of Range».

También hay otros casos en los que se pueden detectar errores, pero por el momento no son muy importantes.

Ahora que ya tenemos en la memoria la rutina en Código Máquina, convendría grabarla (SAVE). En el caso de que surgiera alguna dificultad durante su ejecución, simplemente bastaría con volver a cargarlo. La grabación de una rutina en Código Máquina está muy

bien explicada en el manual del Spectrum pero insistiré ligeramente en este tema.

Todo lo que se necesita es escribir SAVE «Nombre» CODE 30000, 100. Esto le dice al ordenador que guarde o almacene en la cinta cassette el programa en Código Máquina (él ya sabe que se trata de Código Máquina porque has escrito la instrucción CODE seguida de dos números), en las direcciones desde la 30000 hasta la 30000 + 100, con el nombre incluido entre las comillas. Por ejemplo, si hubiéramos querido guardar el anterior programa en Código Máquina, bajo el nombre «pixel-s», entonces habríamos escrito lo siguiente: SAVE «pixel-s» CODE 30000, 100. No te olvides de que sólo estamos guardando el programa en Código Máquina y no el ensamblador o programa en BASIC utilizado para introducir las instrucciones en lenguaje ensamblador dentro del propio programa ensamblador. Cargarlo de nuevo en el ordenador también es muy sencillo. Basta con que escribas: LOAD «pixel-s» CODE. Esto cargará el programa en Código Máquina que tenga el nombre de «pixel-s» en aquellas direcciones en las que se encontraba cuando lo guardamos en la cinta. Una vez finalizada la carga en el ordenador aparecerá el mensaje de error «0» en la parte inferior de la pantalla.

Los desensambladores

Muchas veces nos será de gran utilidad tener la posibilidad de hacer justamente lo contrario de lo que acabamos de hacer. En otras palabras, en lugar de ensamblar una lista de mnemónicos de lenguaje ensamblador a códigos de lenguaje máquina, puede que queramos convertir los códigos de lenguaje máquina en mnemónicos de lenguaje ensamblador. Esto realmente sucede cuando tenemos una rutina en la memoria que no hemos escrito nosotros, o que, habiéndola escrito nosotros, queremos comprobar que todo está correctamente. Otra ventaja de muchos desensambladores es la de que nos mostrarán, no sólo los mnemónicos de lenguaje ensamblador sino también los códigos Hex de dichas instrucciones en lenguaje máquina. Esta posibilidad tiene varios fines, algunos de los cuales no estarán claros hasta más adelante. Yo ya he utilizado ampliamente el desensamblador para realizar las prestaciones finales de los programas que se encuentran en este libro. Utilizar un desensam-

blador es mucho más fácil que utilizar un ensamblador y, por esta razón, voy a dedicar menos tiempo a hablar sobre este tema.

La carga (LOAD) del desensamblador «ACS Software» se realiza de la misma forma que el ensamblador —recordando también cargar la versión adecuada según el ordenador sea de 16K o de 48K . También es importante recordar la posibilidad de tener en la memoria, al mismo tiempo, el ensamblador y el desensamblador en ambas versiones. Esto hace que se convierta en una poderosa herramienta de programación, aun en el caso de que sólo dispongas de un Spectrum de 16K. Esto sólo funciona a condición de que cargues el ensamblador ANTES que el desensamblador. Una vez que se ha cargado el desensamblador en el ordenador ya es posible ejecutar esta rutina a través de uno de los comandos siguientes:

Spectrum de 48K: RANDOMIZE USR 54000.
Spectrum de 16K: RANDOMIZE USR 26600.

Tan pronto como sea ejecutado alguno de estos comandos, entonces aparecerán las palabras STARTING ADDRESS?* (DIRECCION INICIAL?) en la parte superior de la pantalla. Ahora ya sólo es cuestión de introducir la dirección inicial (en decimal) del programa que quieres desensamblar. Si cometes algún error al escribir esta dirección, no utilices «delete». En su lugar pulsa la tecla «E». Tan pronto como hayas introducido con éxito la dirección inicial, aparecerá en la pantalla la primera página con los códigos desensamblados del lenguaje máquina. Si deseas continuar desensamblando escribe la «C»; y si quieres volver a la «Dirección Inicial», entonces escribe «R». Si deseas realizar una copia de lo que se encuentra en la pantalla por medio de la impresora, entonces escribe «P», y si quieres dejar el desensamblador y volver al BASIC, entonces escribe «E».

Una de las características más útiles, tanto del ensamblador como del desensamblador, es que pueden permanecer en la memoria mientras estás utilizando otros programas en BASIC o en Código Máquina pero siempre que el programa en Código Máquina no esté colocado en las mismas direcciones que las que ocupan el ensamblador o el desensamblador y, además, éstos no se verán afectados por la sentencia NEW. Pero ¿es que la sentencia NEW no borra la

* *N. del T.*:Recordamos que el autor del libro se está refiriendo a un programa (de ACS Software) comercializado en Gran Bretaña y, por consiguiente, las instrucciones aparecen en inglés.

memoria en su totalidad? Afortunadamente lo único que hace el comando NEW es volver a poner a cero toda la memoria desde el principio de la RAM hasta el RAMTOP. Solamente hace eso, por lo que cosas tales como los gráficos definibles por el usuario y/o el Código Máquina pueden ponerse a salvo en un sitio en donde no puedan ser afectadas por un programa en BASIC, o por la instrucción NEW.

¿Cómo podemos poner el RAMTOP en una dirección concreta? Basta con escribir CLEAR xxxxx (en donde xxxxx es la dirección del RAMTOP). Esto quiere decir que toda la memoria posterior al RAMTOP quedará protegida. Por ejemplo, para proteger la memoria a partir de 28000 en adelante, de forma que nuestro programa de renumeración, más el ensamblador y el desensamblador, permanezcan a salvo en la memoria, deberíamos escribir: CLEAR 27999. Hay que recordar que el límite del RAMTOP siempre se utiliza, por lo que es necesario poner el RAMTOP en una dirección menos que en la que queremos que empiece la protección.

Te encontrarás con que muchos programas tienen la instrucción CLEAR en su listado BASIC de forma que allí es donde comienza la protección. También es interesante observar que, si te sales del desensamblador, verás que hay una sentencia CLEAR en el breve programa BASIC que ya se encuentra en la memoria. También hay que recordar que aunque CLEAR no borra la pantalla, tampoco borra la memoria por encima del RAMTOP.

Cómo interrumpir el Código Máquina

La manera en la que funciona la tecla BREAK en BASIC es muy sencilla. El ordenador explora el teclado y, si se ha pulsado la tecla BREAK, entonces abandona lo que estuviera haciendo y, al mismo tiempo, escribe un mensaje de error, y luego vuelve al BASIC. Desgraciadamente no podemos hacer eso mismo directamente en Código Máquina debido a que esta facultad no existe en el hardware. Pero podemos resolver este problema utilizando nuestra propia rutina de BREAK (rutina de interrupción), la cual puede ser almacenada junto con las otras rutinas en Código Máquina y que puede ser utilizada cada cierto tiempo con el fin de comprobar si ha sido pulsada la tecla BREAK. Esto es precisamente lo que hace la breve rutina que

damos a continuación. No todas las instrucciones que se utilizan en este programa han sido explicadas por ahora y, de hecho, algunas de ellas, concretamente la instrucción RRA, no serán mencionadas. Por ahora nos es suficiente saber que si se pulsa la tecla BREAK, entonces este programa se parará y volverá al BASIC:

```
LD A, 127
IN A, (254)
RRA
RET NC
```

Por el momento, esta rutina sólo es posible colocarla como una parte de una rutina en Código Máquina pero más tarde aprenderemos cómo hacer que ésta pase a ser una subrutina de nuestra rutina principal. Esto nos proporciona una poderosa y útil herramienta. Más adelante también hablaré acerca de cómo utilizar esta posibilidad, no sólo para detectar si la tecla BREAK ha sido pulsada o no sino también cuando ha sido pulsada cualquier otra tecla. Prosigamos.

Los monitores

Probablemente ya te habrás preguntado a ti mismo ¿qué hacer si no tengo un ensamblador, o no deseo tener uno o, simplemente, no me lo puedo comprar? Desde luego no hay forma de obtener todas las ventajas que ofrece un ensamblador si no compramos dicho programa, pero todavía es posible introducir Código Máquina en la memoria, aunque sólo sea a través de sus códigos en lenguaje máquina, con algunas de las ventajas ofrecidas por un ensamblador —utilizando un MONITOR. También tiene algunas ventajas de las que carecen los ensambladores y que lo compensan un poco, gracias al hecho de que las tablas de conversión al final del libro nos sirven para convertir los mnemónicos del lenguaje ensamblador en códigos de lenguaje máquina, cosa que es totalmente necesaria para que dichos mnemónicos estén ya preparados para poder introducirlos en el «ensamblado a códigos de lenguaje máquina». Dichas tablas de conversión también pueden adquirirse por otros medios.

Los programas correctores (depuradores)

Muchas veces te encontrarás que, aunque hayas comprobado meticulosamente una rutina en Código Máquina, te aparezcan algu-

nos errores (en el argot conocidos por «bugs» o «bichos»). Se cuelan en cualquier parte, ¡hagas lo que hagas! No es posible recuperarse de un fallo en Código Máquina y, por consiguiente, es muy importante que encuentres los errores que tenga el programa antes de que lo ejecutes. Por esta razón, y por otras razones que llegarán a ser más importantes, es una gran ventaja tener un programa corrector. Estos pueden comprarse en cartuchos (es decir, en cartuchos que contienen una cinta cassette) del mismo modo que los ensambladores y los desensambladores. De nuevo, y para tu información, el programa corrector que he utilizado a lo largo de este libro, y que usaré como ejemplo ahora y más adelante, no ha sido producido por la misma casa de software que el ensamblador y el desensamblador, aunque dicha casa también tiene un programa corrector en el momento de publicar este libro. El utilizado aquí está producido por «Artic Computing».

Cargar este programa corrector para el Spectrum resulta tan sencillo como siempre, siendo sólo necesario escribir LOAD »» pero teniendo en cuenta el tipo de ordenador para cargar la versión correcta. Una vez cargado el programa corrector se te ofrecen distintas opciones pero, debido a que este programa ha sido diseñado para ser utilizado con tus propias rutinas en Código Máquina, sería conveniente que, en primer lugar, cargaras en el ordenador la rutina de «scroll por pixel» que acabamos de ver en este mismo capítulo. A pesar de que en dicho programa ya no hay errores (¡ya han sido eliminados!), todavía puede servir como un buen ejemplo. Además, ya que no es posible cargar las rutinas en Código Máquina que no hayan sido grabadas (SAVE) a través de este programa corrector. Primero es necesario salirse del programa corrector y volver de nuevo al BASIC antes de que carguemos el programa (en este caso se trata del programa «scroll por pixel»). Esto se consigue pulsando la tecla «X», seguida de ENTER. Una vez que hayas hecho esto ya puedes cargar la rutina en el ordenador de la forma habitual. Para regresar otra vez al programa corrector del Código Máquina escribe PRINT USR 30884 en el Spectrum de 16K, o PRINT USR 63652 si es que tienes un ordenador de 48K. Es muy importante el hecho de que utilices estos números correctamente (de todas formas vienen reflejados en las instrucciones que acompañan al programa corrector). Fíjate en que en las instrucciones de este programa están lis-

tadas todas las funciones pero éstas están escritas con el fin de tomarlas de referencia, y no para aclarar las cosas. No pretendo examinar todos los comandos y las opciones disponibles de este programa corrector pero vamos a echar una rápida ojeada a los más importantes:

La «Z» nos permite desensamblar una pequeña zona de RAM, al igual que el desensamblador que hemos visto anteriormente, aunque el desensamblador que viene con el programa corrector no es tan perfeccionado. Para desensamblar la zona en la que guardamos nuestra rutina de «scroll por pixel» escribe: Z 7530. Si por alguna razón quieres detener el desensamblaje, pulsa la tecla BREAK. Acuérdate siempre de hacer ENTER una vez que hayas escrito las instrucciones del programa corrector. Suele ser fácil de olvidar.

> *Nota:* Todos los valores que se utilizan en este programa corrector están en hexadecimal, lo que significa que hay un máximo de cuatro cifras, y no están permitidos los números decimales. Por esta razón, puede sernos de gran utilidad volver a los capítulos anteriores y revisar rápidamente cómo se realizan las conversiones entre decimales y hexadecimales. De todas formas, y para que te sirva de ayuda, en el Apéndice A hay una lista de todos los números binarios, decimales y hexadecimales incluidos entre el 0 y el 255 lo que te permitirá hacer dichas conversiones cuando lo necesites. También es posible ejecutar una rutina en Código Máquina desde el programa corrector a través del comando «G» seguido de la dirección, en hexadecimal, en donde comienza la rutina en Código Máquina, seguida de ENTER. Luego también es posible mostrar el contenido de los registros principales pulsando la tecla «D» seguida de ENTER. Esto presentará en la parte superior de la pantalla los valores de todos los registros mayores o fundamentales.

Se trata de una ventaja muy útil si lo hacemos al final del programa pero ¿no nos sería también útil poder conocer el contenido de los registros principales en mitad de la ejecución del programa? Esto puede conseguirse fácilmente utilizando lo que se llama el «punto de ruptura», el cual también es conocido con el nombre de «punto de abandono». Lo que quiere decir es que, cuando el programa

llega a cierta dirección, entonces volverá de nuevo al programa corrector. A partir de ese momento ya puedes presentar en pantalla los registros y, como descubrirás dentro de muy poco, también podrás presentar en pantalla el contenido de los señalizadores. Para establecer un punto de ruptura o de abandono basta con escribir: «Q», seguida de la dirección (en hexadecimal) en la que se quiera poner dicho punto de ruptura. Luego ejecuta la rutina de Código Máquina a través del comando mostrado anteriormente («G»). Tan pronto como la rutina regrese al programa corrector, el punto de ruptura es eliminado.

Ahora prueba a desensamblar el programa, decide dónde vas a poner los puntos de ruptura, colócalos, ejecuta el programa y haz que te muestre los valores de los registros principales en cada punto de ruptura. Mediante la utilización del comando «F» seguido de ENTER, también puedes observar el contenido de los señalizadores. También hay otros comandos menos espectaculares dentro del programa corrector para el Spectrum producido por «Artic» y que vamos a señalar a continuación. Gracias a ellos es posible:

1. Introducir un mensaje en una parte cualquiera de la memoria, sólo con escribir el mensaje por medio del teclado, el cual luego lo convertirá en el valor hexadecimal adecuado.
2. Buscar un bloque de memoria que contenga un valor concreto y mostrarnos la dirección del mismo.
3. Hacer una copia de un bloque de memoria en otro bloque.
4. Mostrarnos los registros de intercambio, que ya hemos visto anteriormente.
5. Hacer que el programa corrector sustituya todos los sucesos acaecidos con un número concreto por los de otro número.
6. Conseguir que el programa corrector cargue y grabe programas escritos por medio de él.
7. Modificar el contenido de un grupo de bytes de la memoria, y también introducir códigos de lenguaje máquina en ciertas direcciones y luego ejecutarlas.
8. Poner en los registros valores concretos. Así podrás simular cierto tipo de hechos dentro del programa.
9. Escribir los códigos de los caracteres BASIC de la rutina y, de esta forma, pueden ser introducidos dentro de una sentencia REM.

¿Qué programa elegir?

Si una vez que llega el momento decisivo resulta que sólo te puedes comprar uno de estos programas, probablemente te convendría más comprar el programa corrector, en primer lugar. No sólo porque te ayuda muchísimo a la hora de encontrar los errores de los programas sino también porque es una gran ayuda para escribir los programas, ayuda a entender cómo funciona el programa y, en general, cómo funciona el ordenador. Si quieres llegar más lejos con el Código Máquina, y deseas escribir programas más largos y más complicados, entonces seguramente el ensamblador será la herramienta más adecuada, sobre todo si consigues emparejarlo con un desensamblador. A continuación ya vendría el programa corrector.

Cuatro instrucciones muy útiles: INC, DEC, RET y NOP

Antes de que escribamos nuestros propios programas, lo cual haremos en el capítulo siguiente, voy a explicar rápidamente cuatro instrucciones muy útiles, las cuales ya hemos utilizado, pero puede que todavía no las hayas entendido del todo, o que simplemente las hemos mencionado pero no las llegamos a explicar. Al igual que todo lo que hemos aprendido hasta ahora, todo se verá más claro una vez que lo pongamos en práctica. Por esta razón, el capítulo siguiente estará exclusivamente dedicado a cómo escribir programas en Código Máquina. Se ha prestado gran atención para lograr una cuidadosa descripción de cada instrucción y del procedimiento necesario para producir buenos programas. Las instrucciones que vamos a ver ahora son: INC, DEC, NOP, RET.

INC

INC es otra de esas instrucciones que tiene dos formas. Puede utilizarse tanto en registros sencillos como en registros dobles. Una de las mayores ventajas de este comando consiste en que puede utilizarse con cualquier registro. INC X hara que el contenido del registro «X» se incremente en uno. Así, por ejemplo, si el registro A contiene el valor 5, y se ejecuta el comando INC A, entonces el

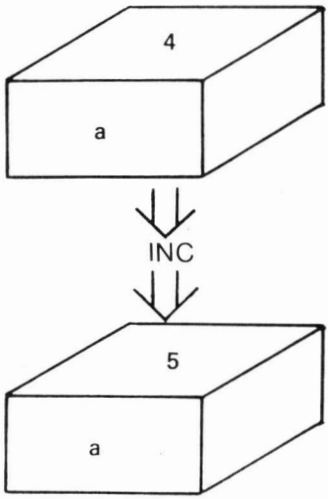


Figura 7

valor del registro A valdrá 6. El efecto será exactamente el mismo ya se trate del registro A o de los registros B, C, D, E, H, o L. En la suma, si el valor de A, o de cualquier otro registro, es 255, y es «INCrementado», entonces el valor del registro se pondrá a cero. Pero hay una pequeña pega. Si se cambia el valor de un registro de 255 a 0 por la instrucción de incremento, el señalizador de acarreo no será marcado (puesto a 1) como sucedería con la instrucción ADD. Este es un factor fundamental que hay que tener en cuenta cuando queramos comprobar, más adelante, el valor de un registro que sea decisivo para el funcionamiento del programa.

La figura 7 es una representación sencilla de lo que hace esta instrucción.

Si hubieras deseado incrementar el contenido de un registro en el lenguaje BASIC, tendrías que utilizar la instrucción LET B = B + 1, en caso de que el registro a incrementar fuera el B. Por medio de este ejemplo resulta evidente que su equivalente en Código Máquina es considerablemente más corto. De ahí el ahorro de memoria y una ejecución mucho más rápida. Los mnemónicos del lenguaje ensamblador y sus códigos Hex equivalentes son los siguientes:

INC A	3C
INC B	04
INC C	0C
INC D	14
INC E	1C
INC H	24
INC L	2C

También es posible el incrementar el valor de un registro doble del mismo modo que lo hacemos con uno sencillo. La única diferencia es que esta instrucción, en ningún caso, afectará a los señalizadores, incluyendo el señalizador de acarreo.

A continuación vamos a ver los mnemónicos del ensamblador y los códigos en hexadecimal de estas instrucciones:

INC BC	03
INC DE	13
INC HL	23

La operación que realiza una instrucción como esta no es muy difícil de comprender aunque puedas tener cierta dificultad la primera vez que lo ves. En caso de que tengas algún problema, he aquí un breve programa en Código Máquina que servirà para ilustrar esta función. Al ejecutar este programa desde el BASIC (utilizando la sentencia PRINT), aparecerá el resultado final del registro doble BC. Sería interesante que modificaras varias veces el contenido del registro BC de la primera sentencia con el fin de observar cómo cambia la respuesta.

```
ORG 30000
30000 01 00 00      LD BC,0000
30003 03            INC BC
30004 C9            RET
```

El incremento de una posición concreta de la memoria

Todavía hay una función de la sentencia de INCremento de la que no hemos hablado. Se trata de la posibilidad de incrementar el contenido de una posición concreta de la memoria. Por ejemplo,

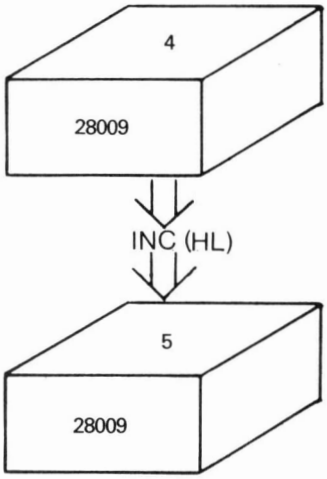


Figura 8

si primero cargas en el registro doble HL la dirección de una posición cuyo contenido va a ser incrementado, al ejecutar la instrucción «INC (HL)» se incrementará en uno el contenido de la dirección HL, pero EL VALOR DEL PROPIO HL NO SERA CAMBIADO. La figura 8 nos lo muestra un poco más claramente.

El código hexadecimal en lenguaje máquina de esta instrucción es 34. Pero acuérdate de que esta instrucción sólo puede usarse en unión del registro HL.

Ahora introduce el programa listado a continuación, ejecútalo y luego comprueba el valor de la posición 28012. Vuelve a ejecutar la rutina y comprueba de nuevo el contenido de la posición 28012. Descubrirás que se ha incrementado en 1. Fíjate en que la operación de incrementar el valor de una posición sólo afecta a esa posición concreta. De ahí que el máximo valor al que puede llegarse es 255 que, como ya sabes, es el contenido máximo de una posición.

He aquí el programa:

```
ORG 30000
30000 21 6C 6D        LD HL,28012
30003 23              INC(HL)
30004 C9              RET
```

DEC

Esta instrucción funciona exactamente igual que INC, excepto que, en lugar de incrementar, lo que hace es disminuir (o DECrementar) el valor de un registro o de una posición. ¡Va hacia abajo en vez de ir hacia arriba! Por ejemplo, si decidimos ejecutar el comando DEC A cuando A ya valga 7, entonces el resultado será igual al valor de A menos 1, por lo tanto sería 6. La figura 9 nos muestra claramente este proceso.

A continuación hay una lista de los mnemónicos del lenguaje ensamblador y los códigos hexadecimales del lenguaje máquina de estas instrucciones:

DEC A	3D
DEC B	05
DEC C	0D
DEC D	15
DEC E	1D
DEC H	25
DEC L	2D

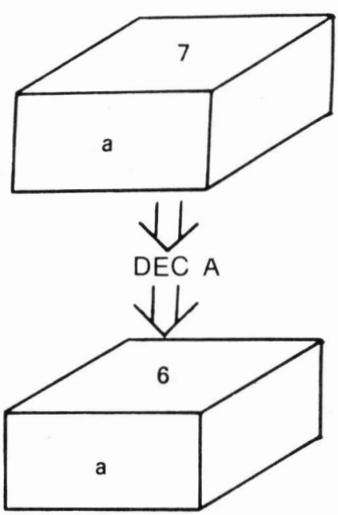


Figura 9

A estas alturas ya deberías ser capaz de aprovechar el programa ejemplo utilizado anteriormente para probar ambos comandos INC y DEC. En caso de que todavía encuentres esto un tanto confuso, aquí tienes la rutina convenientemente arreglada:

```
ORG 30000
30000 01 00 00      LD BC,0000
30003 0B            DEC BC
30004 C9            RET
```

De nuevo, el señalizador de acarreo no queda afectado por la ejecución de la instrucción de decremento, tanto con un registro sencillo como con un registro doble. Los mnemónicos del lenguaje ensamblador y los códigos hexadecimales de las instrucciones de decremento utilizadas en los registros dobles se muestran a continuación:

DEC BC	0B
DEC DE	1B
DEC HL	2B

Como puedes observar, por los mnemónicos y por los códigos hexadecimales anteriores, los códigos Hex guardan algún tipo de relación con los mnemónicos del lenguaje ensamblador. Por ejemplo, todas las instrucciones de disminución de los registros dobles tienen una B en el segundo dígito. Es muy útil acordarse de pequeños trucos como este —sobre todo si no tienes un ensamblador. Dedica algunos minutos a ojear el libro para ver si puedes encontrar algún caso de relación similar.

Al igual que en la instrucción de incremento, también es posible disminuir el contenido de una posición concreta cuando la dirección de esta posición está almacenada en el registro doble HL. He aquí un ejemplo de lo que acabo de decir:

```
ORG 30000
30000 21 6C 6D      LD HL,28012
30003 2B            DEC(HL)
30004 C9            RET
```

Después de ejecutar esta rutina, comprueba el valor de la posición 28012 haciendo un PEEK desde el BASIC. Luego vuelve a eje-

cutarla y comprueba una vez más el contenido de la posición 28012. En cualquier caso, excepto si el valor previo de esa posición fuera 0, el contenido habrá disminuido en 1. Si, por otro lado, el contenido de esa posición ya es 0, entonces, cuando se le reste 1, se convertirá en 255.

RET

A estas alturas me imagino que casi con toda certeza habrás comprendido que esta instrucción provoca la finalización de la rutina, y el control del ordenador vuelve al BASIC. Siempre es necesario que haya una instrucción RET al final del programa, a no ser que utilices una de sus pequeñas variantes que manejaremos más adelante. Estas por el momento no tienen nada que ver con lo que estamos hablando. El código Hex para el mnemónico de lenguaje ensamblador «RET» es muy fácil de recordar y probablemente, en las próximas páginas se te hará muy familiar —es C9. La razón de esta familiaridad viene dada por su uso continuo.

NOP

Se trata de un comando muy fácil de entender. No hace absolutamente NADA. Cuando el ordenador llega a esta instrucción, cuyo códigos es 0, no hará absolutamente nada, y continuará ejecutando la instrucción siguiente, dejando tal y como estaban el contenido de todos los registros, todas las direcciones y todos los señalizadores. Puede parecer un tanto extraño que alguien quiera incluir una instrucción que, aparentemente, no hace nada, cuando resulta que es muy útil. Esto es así porque, a menudo, es importante dejar zonas de una rutina en las que no se haga nada. Esto quiere decir que, más tarde, tendrás la posibilidad de incluir nuevos comandos. Esta práctica es parecida a la manera en que, en un programa BASIC, dejas libres unas cuantas líneas. En otras palabras, nunca sueles numerar las líneas de programa con 1, 2, 3, 4, ya que no dejarías sitio para poder maniobrar. De forma parecida, puede sernos de mayor utilidad tener un comando disponible que nos asegure, cuando nos haga falta, que el ordenador no pueda operar.

Hasta ahora nos hemos concentrado bastante en la parte teórica

del Código Máquina, aunque ya hemos visto varios ejemplos y hemos incluido varios programas para que los ejecutes. De cualquier forma la mayor atracción es el utilizar el Código Máquina. El capítulo siguiente, por lo tanto, estará centrado en el proceso de «creación» de un programa, su codificación y, finalmente, su corrección, ejecución y modo de guardarlo en una cinta. Antes de finalizar este capítulo vamos a ver un programa complementario de uno anterior. Este hace el «scroll por pixel hacia arriba». Puedes utilizar tanto un ensamblador, el monitor de BASIC o el programa corrector.

¡Diviértete!

```
ORG 30000
30000 21 00 40      LD HL,16384
30003 E5            PUSH HL
30004 11 00 5B      LD DE,23296
30007 01 20 00      LD BC,32
30010 ED B0         LDIR
30012 E1            POP HL
HL=DIRECCION MAS ALTA DEL BLOQUE
   SUPERIOR
B=NUMERO DE BLOQUES
30013 06 03         LD B,3
A
30015 C5            PUSH BC
30016 E5            PUSH HL
30017 06 08         LD B,8
B
30019 C5            PUSH BC
30020 E5            PUSH HL
30021 06 07         LD B,7
C
30023 C5            PUSH BC


30024 E5            PUSH HL
30025 E5            PUSH HL
30026 D1            POP DE
30027 24            INC H
30028 01 20 00      LD BC,32
30031 ED B0         LDIR
```

```
30033 E1            POP HL
30034 24            INC H
30035 C1            POP BC
30036 10 F1         DJNZ,C
30038 E5            PUSH HL
30039 11 E0 06      LD DE,1760
30042 ED 52         SBC HL,DE
30044 D1            POP DE
30045 01 20 00      LD BC,32
30048 ED B0         LDIR
30050 E1            POP HL
30051 11 20 00      LD DE,32
30054 19            ADD HL,DE
30055 C1            POP BC

30056 10 D9         DJNZ,B

30058 11 E0 06      LD DE,1760
30061 19            ADD HL,DE
30062 E5            PUSH HL
30063 11 20 00      LD DE,32
30066 19            ADD HL,DE
30067 D1            POP DE
30068 01 20 00      LD BC,32
30071 ED B0         LDIR
30073 E1            POP HL
30074 11 00 08      LD DE,2048
30077 19            ADD HL,DE
30078 C1            POP BC
30079 10 BE         DJNZ,A
30081 11 20 00      LD DE,32
30084 ED 52         SBC HL,DE
30086 EB            EX DE,HL
30087 21 00 5B      LD HL,23296
30090 06 20         LD B,32
D

30092 7E            LD A,(HL)
30093 00            NOP
LD A,0 PARA EVITAR BUCLE SIN FIN
30094 12            LD (DE),A
```

```
30095 13          INC DE
30096 23          INC HL
30097 10 F9       DJNZ,D
30099 C9          RET
```

¿Sabes cómo utilizarlo?

Capítulo 7

LA CONFECCION DE UN PROGRAMA

Ya hemos progresado bastante en el aprendizaje sobre cómo utilizar el Código Máquina y las instrucciones correspondientes. Ahora es un buen momento para empezar a ver cómo podemos realizar nuestros propios programas en Código Máquina. Gracias a unos cuantos comandos lo que va a continuación se encuentra dentro de nuestros conocimientos y, por lo tanto, podemos hacer buen uso de ello:

1. Sumar registros.
2. Sumar a los registros.
3. Restar un registro de otro.
4. Restar números a los registros.
5. Incrementar un registro.
6. Incrementar el valor de una posición.
7. Decrementar o disminuir un registro.
8. Decrementar o disminuir el valor de una posición.
9. Cargar un registro con un número determinado.
10. Cargar un registro con el valor de otro registro.
11. Cargar un registro con el contenido de una posición.
12. Cargar una posición con el valor de un registro.
13. Regresar al BASIC y no hacer absolutamente nada.

Con todo este material ya podemos componer una rutina en Código Máquina bastante decente. De todas formas ahora es cuando

te darás cuenta de que no es tan fácil como programar en BASIC. Pero debes perseverar si quieres cosechar buenos resultados. No basta con sentarte delante del teclado y programar. En su lugar, debes actuar de forma lógica y sistemática. Debes decidir en cada caso qué es lo que quieres hacer y cómo lo quieres hacer. Asegúrate siempre de que no has cometido ningún error ya que no es posible recuperarse de un fallo en Código Máquina. Otro factor importante es recordar que no es tan fácil cambiar un programa en Código Máquina como uno escrito en BASIC. Por consiguiente sería de una ayuda inestimable tener diagramas de flujo y anotaciones sobre cómo has escrito exactamente tu programa. Lo que sigue a continuación es un esquema del proceso de «organización» y de la estructura de los programas. Llegarás a tener tu propia manera personal de hacer este tipo de cosas, pero, por ahora, aprovéchate estudiando la mía:

1. RESUMEN DE LA IDEA

Decide con precisión cuál es el objetivo del programa o de la rutina una vez que estén terminados. Esto requerirá que estudies cuidadosamente el problema que deseas resolver o las ideas que quieres poner en práctica, y luego decidir exactamente qué es lo que quieres que haga el ordenador. Escríbelo y utilízalo como referencia a medida que avanzas en su construcción.

2. ESQUEMA DEL ORGANIGRAMA O DIAGRAMA DE FLUJO

Esto te sirve para desglosar la idea general en partes más pequeñas que lo componen. Todo te será mucho más fácil si trabajas por secciones separadas. A estas alturas todavía no es necesario llegar a un nivel muy detallado de lo que quieres hacer en cada una de dichas secciones. Basta con que pongas el orden en que quieres que las secciones de tu idea sean ejecutadas.

3. ELABORACIÓN DE LA ESTRUCTURA

Esto trae consigo el examen de todo el programa, sección por sección, analizando qué operación o movimiento quieres que realice el ordenador en cada momento. Recuerda siempre el grupo de comandos que se encuentran a tu disposición. De aquí nos vamos al paso siguiente dentro de este proceso de análisis.

4. EL DIAGRAMA DE FLUJO DEFINITIVO

Básicamente se trata de una mezcla de todas las secciones de la rutina, colocadas secuencial y lógicamente de manera que sea fácil entenderlas.

5. CONVERSIÓN

Esto no quiere decir que haya que cambiar el lenguaje ensamblador a Código Máquina sino que se trata de una operación básica que consiste en transformar lo que hayas programado en forma de breves sentencias pasándolas a instrucciones de lenguaje ensamblador. Estas pueden colocarse en versión «ensamblador», es decir, en mnemónicos. Ahora pon las instrucciones en un orden lógico tal y como quieras que las ejecute el ordenador.

6. EJECUCIÓN «EN SECO»

Ahora ejecuta el programa, paso a paso, no utilizando el ordenador sino con papel y bolígrafo, anotando el contenido de los registros que vas utilizando, el contenido de aquellas direcciones que van a ser modificadas, y el estado en que se encuentran los señalizadores (las banderas). Si todo marcha bien es que estás preparado para introducirlo en el ensamblador, pero si no, por lo menos no has perdido el tiempo convirtiéndolo en Código Máquina, tecleándolo y ejecutándolo antes de descubrir un posible error.

7. INTRODUCCIÓN DEL PROGRAMA EN EL ORDENADOR

Es muy importante que ya estés familiarizado con el procedimiento que utilice tu ensamblador y este es el motivo por el que he decidido permanecer con el mismo programa a lo largo del libro.

8. COMPROBACIÓN

Antes de continuar vale la pena hacer una comprobación, bien con un desensamblador, bien a través de un examen, con objeto de asegurarte de que no has cometido algún error al introducir el programa con el ensamblador. Muchas veces los errores son detectados por el ensamblador pero también sucede que otras veces no y eso puede provocar el ensamblaje de una instrucción completamente diferente. Esto puede significar el derrumbamiento de todo el programa sin que sepamos por qué se ha producido.

9. DEPURACIÓN DE ERRORES

Podemos servirnos de un programa depurador cuya utilización ya ha sido explicada. La corrección de un programa mientras se encuentra en la memoria no sólo es de gran utilidad para comprobar los errores dentro del programa sino que también ayuda a entender mucho mejor qué es lo que sucede dentro del ordenador.

10. GRABACIÓN EN CINTA CASSETTE

Como ya he señalado, la grabación del Código Máquina es una operación distinta a la de la grabación de un programa BASIC. Por lo tanto, es importante poder grabar los dos, así como poder almacenar en un soporte permanente las rutinas.

11. LA EJECUCIÓN

Ahora te encuentras en una situación en que todo está «preparado» (O.K.) para ejecutar la rutina. Aun así no se trata de la operación más sencilla de realizar.

Ahora vamos a comenzar este proceso con nuestra propia idea. Así podremos ir siguiéndola de cerca hasta llegar al programa final, que veremos al término de este capítulo.

La idea

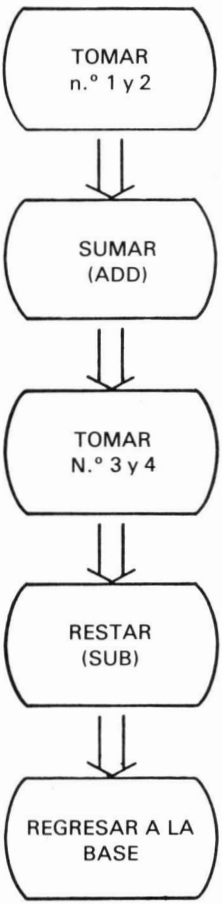
Lo que vamos a hacer es escribir una rutina que:

1. Sea accesible desde el BASIC.
2. Sea capaz de comunicarse con el BASIC.
3. Pueda sumar dos números de 16 bits.
4. Pueda restar un número de 16 bits de otro, dejando el resultado disponible para ser utilizado por el programa BASIC.

Elaboración de la estructura

El primer problema con el que nos tenemos que enfrentar es cómo asignar los valores desde el BASIC al programa en Código Máquina, de forma que los números de 16 bits que vayan a ser sumados o restados puedan ser controlados a través del BASIC. Recuerda que no podemos asignar una variable en BASIC y esperar

El esquema del diagrama de flujo (fig. 10)



que sea reconocida por el Código Máquina. La transferencia del contenido de una variable BASIC a un registro del Código Máquina es un procedimiento muy complicado. Por esta razón creo que esto no tendría una solución viable. Recuerda también que no es posible cargar un registro desde el BASIC ya que en el BASIC no existen instrucciones para manipular dichos registros. Afortunadamente hay una operación que puede ser realizada a través del BASIC o a través del Código Máquina. Se trata de transferir un número a una posición concreta de la memoria y luego transferir ese valor fuera de dicha posición bien a un registro, bien a una variable como se realiza en BASIC. Si queremos cargar un número en una posición a través del BASIC entonces utilizaremos el comando POKE. Para transferirlo de esa posición a un registro utilizaremos la instrucción "LOAD registro, (posición)". Lo que realmente estamos haciendo no es más que colocar un número en una caja, ir al Código Máquina y luego volver a quitarlo de allí.

El problema siguiente consiste en trasladar las respuestas desde el Código Máquina a una forma accesible a través del BASIC. En este caso hay dos métodos diferentes de resolver este problema. Debido a lo que acabamos de decir un poco más arriba, parecería totalmente infructuoso investigar la posibilidad de acceder a un registro del Código Máquina desde el BASIC.

La primera solución puede ser utilizada de forma similar al modo en que en primera instancia la trasladábamos al programa en Código Máquina. Con otras palabras, cargar una posición de la memoria con el resultado mientras todavía está en Código Máquina y luego volver a tomar el resultado cuando nos encontremos de nuevo en el BASIC.

Por otro lado, si se utiliza la instrucción "PRINT USR xxxxx", en donde xxxxx es la posición del programa en Código Máquina, entonces se escribirá el contenido del registro doble BC cuando se termine la rutina en Código Máquina. Esto quiere decir que, si nos aseguramos de que el resultado va a estar en el registro doble BC, entonces lo único que tendremos que hacer es utilizar la sentencia PRINT. También funcionaría de igual modo si utilizáramos la instrucción «LET A = USR xxxxx», en donde de nuevo xxxxx es la dirección del programa en Código Máquina, ya que el contenido del registro doble BC será trasladado a la variable A. Evidentemente esta solución es muy útil pero su mayor inconveniente es que sólo puedes almacenar el re-

sultado en el registro doble BC. En lugar de regresar al BASIC y luego tener que volver a introducir el Código Máquina, es conveniente utilizar una combinación de las dos soluciones. En otras palabras, deberemos almacenar el primer resultado en direcciones accesibles a través del BASIC, y tener el segundo resultado almacenado en BC. De esta forma podrá escribirse o ser trasladado a una variable.

Una vez que hayamos decidido exactamente cómo vamos a trasladar las constantes desde el BASIC al Código Máquina y el modo en que vamos a trasladar los resultados desde el Código Máquina otra vez al BASIC, debemos resolver la secuencia de sucesos entre estas dos operaciones. Lo que vamos a decidir ahora es si vamos a hacer primero la rutina de suma o la de resta. Se trata de algo totalmente indiferente, siempre y cuando recordemos poner el resultado de la primera rutina en una posición de memoria y el resultado de la segunda rutina en el registro doble. En este caso he decidido realizar primero la suma antes que la resta. Esta decisión no la he tomado después de un excesivo proceso analítico. Simplemente lanzando una moneda al aire.

Diagrama de flujo (2.ª versión)

A continuación tenemos la segunda versión del diagrama de flujo original. Obsérvalo cuidadosamente a la vista de lo que acabamos de decir:

1. Toma los dos primeros valores de sus direcciones de la memoria.
2. Suma los dos números.
3. Almacena el resultado en una caja, o posición de memoria.
4. Toma los segundos valores de sus respectivas posiciones de memoria.
5. Resta uno del otro.
6. Almacena el resultado en el registro doble BC.
7. Regresa al BASIC.

Ahora, en el BASIC

1. Almacena los cuatro valores.
2. Ejecuta la rutina en Código Máquina.

3. Escribe el resultado de la segunda rutina.
4. Recupera y presenta el resultado de las primeras rutinas que cargamos anteriormente en las posiciones de memoria a través de la rutina de Código Máquina.
5. Fin.

¿Cómo sumamos los números de 16 bits?

Vamos a suponer que los números tomados de la memoria del ordenador han sido almacenadas en los registros HL y DE. El resultado de la suma deberá estar almacenado, por tanto, en el registro doble HL. Es importante recordar que H es el byte más significativo de HL, y L es el byte más bajo (o byte menos significativo) de HL. Del mismo modo, D es el byte más alto (o más significativo) del registro doble DE, y E el byte más bajo. Como en la suma ordinaria, primero sumamos los «bytes más bajos». Hacer otra cosa sería mucho más problemático. Esto se pondrá más de manifiesto a medida que avances.

Por consiguiente, necesitamos sumar el registro E al registro L. Desgraciadamente, como debes recordar, sólo es posible sumar otros registros al registro A. Así que ¿cómo resolvemos esto? Muy sen-

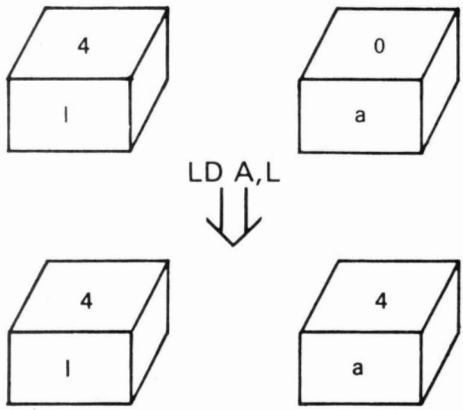


Figura 11

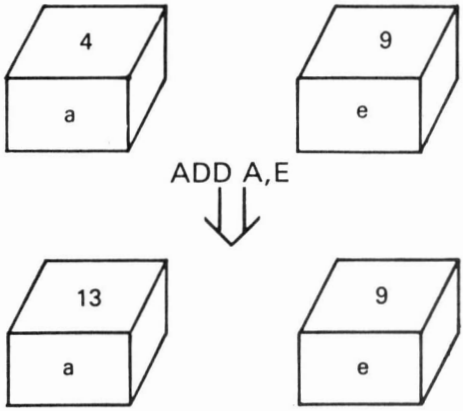


Figura 12

cillo. Lo único que tenemos que hacer es trasladar el valor de L al registro A, luego sumar el registro E al registro A. El resultado de la suma de E con L ahora está guardado en el registro A, de forma que la respuesta que está en A puede ser trasladada de nuevo al registro L.

Ya hemos decidido cuáles van a ser las dos primeras instrucciones de nuestro programa en Código Máquina. Primero debemos «trasladar L a A», como se muestra en la figura 11.

Luego tenemos que sumar el registro E al registro A. De nuevo esto se muestra en la figura 12.

Ahora ya tenemos el resultado en el registro A, pero ¡¡eso no era lo que queríamos!! Lo que queremos es que el resultado completo esté en el registro doble HL, por lo que tenemos que poner el resultado de la suma de los bytes más bajos en el registro L. Esto puede hacerse por el simple traslado del contenido de A al registro L (Fig. 13).

Ahora tenemos que echar una ojeada a lo que sucede cuando, al sumar dos bytes, el resultado es un número mayor que 255 ya que 255 es el mayor número que puede ser guardado en un registro sencillo. Una buena cosa que tienen los microprocesadores es que la respuesta a este hecho es totalmente consistente y pronosticable. ¡Todo se pone a cero! Al sumar 1 a 255 vuelve a empezar

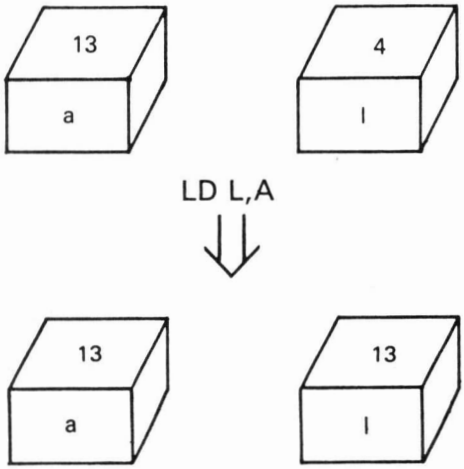


Figura 13

en 0; al sumar 20 a 255 lo vuelve a 0 y luego llega hasta 19. Adicionalmente, el señalizador de acarreo es marcado con un 1. El señalizador de acarreo es un bit dentro del registro F al que se remite la propia CPU con objeto de averiguar cuándo se ha desbordado un número o no. Si un número ha ido más allá de la barrera de 255, entonces definitivamente se habrá producido un desbordamiento, y el señalizador de acarreo marcará un 1. Si, por supuesto, no existe desbordamiento, entonces el señalizador de acarreo se pondrá a 0. Es absolutamente esencial referirse a este punto de vez en cuando. Su importancia debe estar siempre presente.

El señalizador de acarreo es un bicho raro: no se puede llegar a él directamente y por eso no es posible cargar A, por ejemplo, con el estado en que se encuentra dicho señalizador de acarreo. Por otro lado, hay instrucciones que tienen en cuenta al señalizador de acarreo. No obstante. el estado del señalizador de acarreo puede ser controlado y por eso puede ser marcado y puesto a cero a voluntad. Tendremos que utilizar uno de esos comandos que tienen en cuenta al señalizador de acarreo cuando sumemos los bytes más altos. Tendremos que utilizar una instrucción que haga lo siguiente: sumar E con H más el estado del señalizador de acarreo. El estado del señalizador de acarreo se establece al hacer la suma de los bytes menos

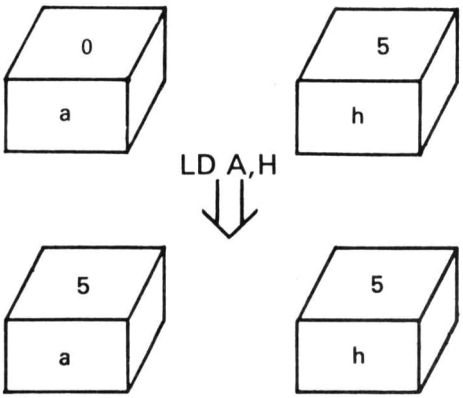


Figura 14

significativos y, si no se cambia por ninguna otra operación con los registros, entonces se producirá la suma correcta.

Volvamos a la tarea que tenemos entre manos. Nos encontrábamos en el momento en que acabábamos de sumar los bytes menos significativos y estábamos a punto de sumar los bytes más significativos. Dado que no podemos sumar D con H, sino D a A, trasladamos H a A. Ejemplo (figura 14).

Y ahora sumamos el registro D al registro A y tomamos en cuenta el estado del señalizador de acarreo. Verbigracia (fig. 15):

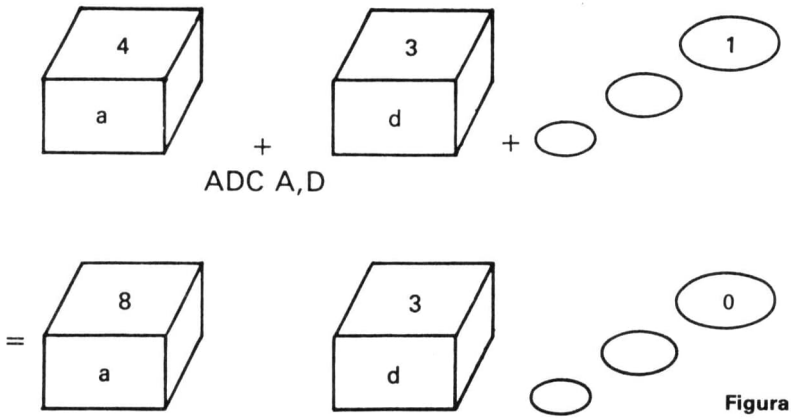


Figura 15

La instrucción siguiente será «sumar D y el acarreo al registro A».

Ahora ya tenemos el resultado de D más H, más el señalizador de acarreo en el registro A, y queremos volver a tenerlo en el registro H de forma que el resultado quede en HL. El traslado se efectúa de la forma que muestra la figura 16.

«AHORA EL RESULTADO YA LO TENEMOS EN HL (¡Por fin!)»

La suma ya está completa pero ahora tenemos que hacerla accesible a través del BASIC. También podríamos poner el valor de HL en BC, de forma que pueda ser escrito en la pantalla, o que sea trasladado a una variable Basic, o almacenar el resultado en posiciones de la memoria. Todas estas opciones pueden ser alcanzadas a través del BASIC. De todas estas posibilidades, la de utilizar BC para guardar el resultado parecería ser la que encajara mejor dentro de nuestros objetivos. De cualquier forma, como lo más probable es que hagamos uso del registro BC durante la sección del programa dedicada a la resta, tendremos que utilizar las posiciones de memoria para el almacenamiento. Para la suma de estos dos bytes se necesitan dos posiciones de memoria (16 bits). Las posiciones en las que se va a colocar el resultado pueden encontrarse en cualquier parte de la RAM pero primero asegúrate de que no le vas a superponer otro programa. Si tienes alguna duda, entonces vuelve a repasar la sección en la que aparece el mapa de memoria. Para nuestros propósitos las direcciones 27004 y 27005 serán más que suficientes.

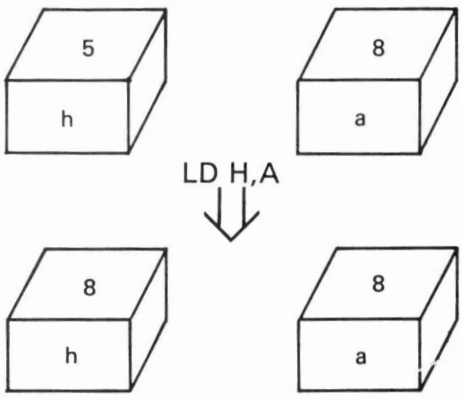


Figura 16

Es posible dividir el resultado y ponerlo en dos direcciones que no sean adyacentes pero no tendría mucho sentido y haría que la programación fuera más difícil. Ahora podemos indicar al ordenador que almacene HL en las posiciones 27004 y 27005.

> *NOTA:* El ordenador almacena el byte menos significativo antes que el byte más significativo por lo que debes acordarte de multiplicar el contenido de la segunda dirección por 256, en lugar de la primera.

Ahora tenemos que restar un registro de 16 bits de otro, utilizando un comando de resta de registros dobles. Pero ¿cómo? Toma como punto de partida que los dos valores que necesitamos han sido introducidos en los registros dobles DE y HL. El objetivo consiste en restar el valor de DE al valor de HL. El resultado quedará en HL. Para conseguir esto sólo se necesita un comando pero recuerda que este comando también afecta al señalizador de acarreo. Si la última instrucción ejecutada ha hecho que el estado del señalizador de acarreo se haya puesto a 1, entonces nuestro resultado puede ser erróneo, por lo que tenemos que asegurarnos de que el estado del señalizador de acarreo es 0. Esto se puede conseguir indirectamente. Por el momento vamos a hacerlo con esta instrucción. «Pon el señalizador de acarreo a 0», por ejemplo (fig. 17):

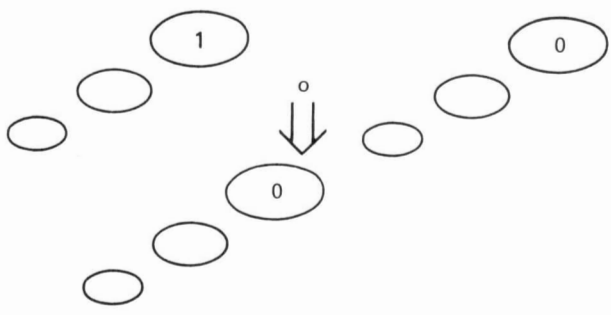


Figura 17

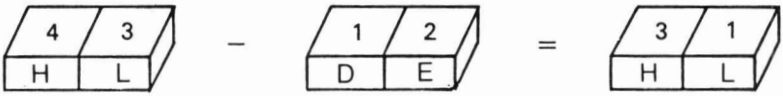


Figura 18

Con el señalizador de acarreo en 0 el resultado de la resta no se verá afectado. Podemos seguir adelante y restar (con acarreo) DE a HL —mostrado gráficamente en la figura 18.

Efectivamente le has restado a HL el valor de DE, dejando el resultado en HL. Ahora lo único que hay que hacer es decidir cómo trasladar el resultado a alguna forma a la que se pueda acceder a través del BASIC. Anteriormente habíamos evitado utilizar el registro doble BC para el caso de que lo necesitáramos durante la construcción del programa. Quizás sea una forma muy laboriosa de lograr lo que queremos pero, sin embargo, vale la pena. Ahora que ya es evidente que «BC» está libre para ser utilizado, vamos a apropiarnos de él. Traslada H a B, y traslada L a C. El resultado, que originariamente estaba guardado en HL, ahora estará guardado en BC. Véase la figura 19.

La forma en la que funciona nuestro programa ya se encuentra bajo control por lo que el siguiente paso consiste en trasladar los valores originales del BASIC a la memoria y luego dejar que el Código Máquina realice el traslado del contenido de cada posición a los registros dobles correspondientes.

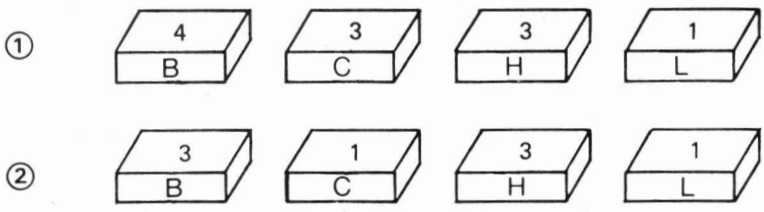


Figura 19

Traslado de los valores iniciales del BASIC para su uso en Código Máquina

El primer requisito consiste en convertir los cuatro números de 16 bits en parejas de números de 8 bits. Estos pueden ser colocados en la memoria y luego ser llamados por el Código Máquina. Este proceso es relativamente fácil trabajando en hexadecimal. Toma las dos primeras cifras y averigua su equivalente en "base 256". Luego toma las dos últimas cifras y halla su equivalente en "base 256". El resultado nos dará las dos mitades del número de 16 bits. Al final del libro encontrarás una tabla con todos los números desde el 0 al 255 en binario, hexadecimal y decimal. No dudes en mirar las tablas. Nos resolverán parte del trabajo relativo a este ejercicio. Recuerda que el ordenador por sí mismo guardará primero el byte más bajo (menos significativo) y luego el byte más alto (más significativo). Haz lo mismo. Mientras que no se altere el resultado, siempre y cuando no se realicen cambios importantes en el programa, te ayudará a "adquirir una buena costumbre".

Los números de 8 bits pueden ser almacenados en cualquier parte de la RAM, siempre y cuando no interfieran ninguna otra cosa. Yo he elegido utilizar la dirección 27000 como punto inicial. Para la suma:

L = 27000
H = 27001
E = 27002
D = 27003

y para la resta:

L = 27006
H = 27007
E = 27008
D = 27009

El resultado de la suma estará guardado en las direcciones 27004 y 27005.

Una vez convertidos los números de 16 bits en números de 8 bits resulta un trabajo muy fácil colocar estos números en las direcciones preparadas para la ejecución de la rutina de Código Máquina.

¿Cómo nos introducimos en el Código Máquina?

No es posible GOTO (ir a) la rutina de Código Máquina o GOSUB (ir a la subrutina) ya que ambas instrucciones son para el uso de programas en BASIC. Pero el BASIC tiene un comando especial para acceder al Código Máquina del usuario —USR— que significa **U**ser **S**ub **R**outine (SubRutina del Usuario). Dentro del contexto del BASIC, USR no es un comando completo. Por eso es necesario prefijarlo con uno de los comandos siguientes:

PRINT. De aquí que se escriba en la pantalla el valor de regreso del registro doble BC.

LET, que quiere decir que puedes asignar a una variable el valor contenido en el registro doble BC al regresar del Código Máquina, y RANDOMIZE, el cual pone en el origen aleatorio (RND) el resultado del registro doble BC.

Tanto PRINT como LET son un gran desperdicio de memoria y, por esta razón, generalmente se utiliza RANDOMIZE como comando para acceder a la rutina USR. Una rutina USR se parece mucho a una rutina GOSUB en que ambas son subrutinas. Para salirse de ellas hay que incluir la instrucción RETURN, que en el contexto del Código Máquina, ha sido acortada a RET (RETorno o regreso).

A continuación debemos ver la manera de acceder a las constantes que han sido almacenadas a través del BASIC. La mejor manera de representar a los números almacenados en sus posiciones de memoria es por medio de un dibujo. (Ver figura 20)

Ante esto, quizás la siguiente operación sería utilizar un comando para cargar el contenido de la dirección 27000, por ejemplo, en el registro C. Ten la seguridad de que esto acarrearía la utilización de

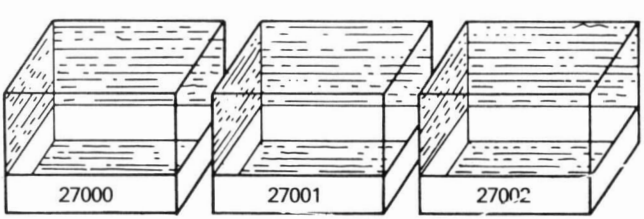


Figura 20

rutinas demasiado largas y, por lo tanto, sería bastante descorazonador. En su lugar vamos a hacer uso de la posibilidad de cargar un registro con el contenido de una posición y cuya dirección está guardada en HL. Utilizando este método, e incrementando o disminuyendo el valor de HL para seleccionar los bytes que hay que cargar en los registros, haremos algo que se conoce con el nombre de "utilización del registro HL como un puntero". La operación que realiza un puntero se clarifica en el dibujo de la figura 21.

Ahora vamos a ver cómo funciona todo esto:

- Primero pon el puntero en 27000; "LD HL, 27000".
- Toma la primera mitad del primer número de 16 bits y ponla en C; "traslada el contenido de la posición HL a C".
- Mueve el puntero una posición; "incrementa en uno el valor de HL".
- Pon la segunda mitad del primer número de 16 bits en B; "traslada el contenido de HL a B".
- Mueve el puntero una posición; "incrementa en uno el valor de HL".
- Traslada la primera mitad del segundo número de 16 bits a E; "traslada el contenido de HL a E".
- Mueve el puntero una posición; "incrementa en uno el valor de HL".

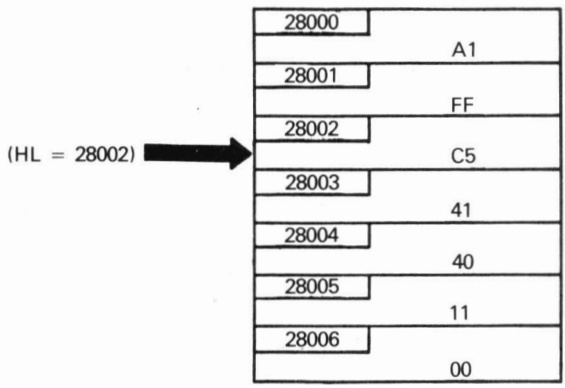


Figura 21

—Traslada la segunda mitad del segundo número de 16 bits a D; "traslada el contenido de la posición HL a D".

Ahora la transferencia ya está completa pero queda un problema y es que tenemos que cargar uno de los valores en BC ya que HL fue utilizado como puntero. HL es el único registro doble que puede ser utilizado como puntero, del mismo modo que es el único registro que puede guardar una dirección a la que se pueda acceder. Haz las dos instrucciones siguientes: "traslada B a H" y "traslada C a L". Las constantes que habían sido almacenadas por el BASIC ahora se encuentran en sus correspondientes registros dobles del Código Máquina.

El paso siguiente consiste en el traslado de las constantes del BASIC a Código Máquina para la sección de la resta. Fíjate en las grandes ventajas del sistema del puntero. Una de las mejores cosas de este sistema es la facilidad con que el ordenador llega a las distintas direcciones, simplemente cambiando el contenido inicial del registro HL —que actúa como puntero. Otro aspecto del sistema de puntero es que es posible no sólo acceder a una posición después de la otra, yendo hacia adelante, sino también acceder a una posición tras otra en sentido contrario (disminuyendo el valor del puntero). Esto se verá más claro en el desarrollo, paso a paso, de cómo poner los valores de la memoria en las posiciones de los registros (fig. 22).

— Carga el puntero con valor más alto: "LD HL, 27009".
— Traslada la segunda mitad del segundo número de 16 bits a D: "traslada el contenido de la posición HL a D".
— Mueve el puntero una posición hacia atrás: "disminuye el valor de HL en 1".
— Traslada la primera mitad del segundo número de 16 bits a E: "traslada el contenido de la posición HL a E".
— Mueve el puntero una posición hacia atrás: "disminuye en 1 el valor de HL".
— Traslada la segunda mitad del primer número de 16 bits a B: "traslada el contenido de la posición HL a B".
— Mueve el puntero una posición hacia atrás: "disminuye en 1 el valor de HL".
— Traslada la primera mitad del primer número de 16 bits a C: "traslada el contenido de la posición HL a C".

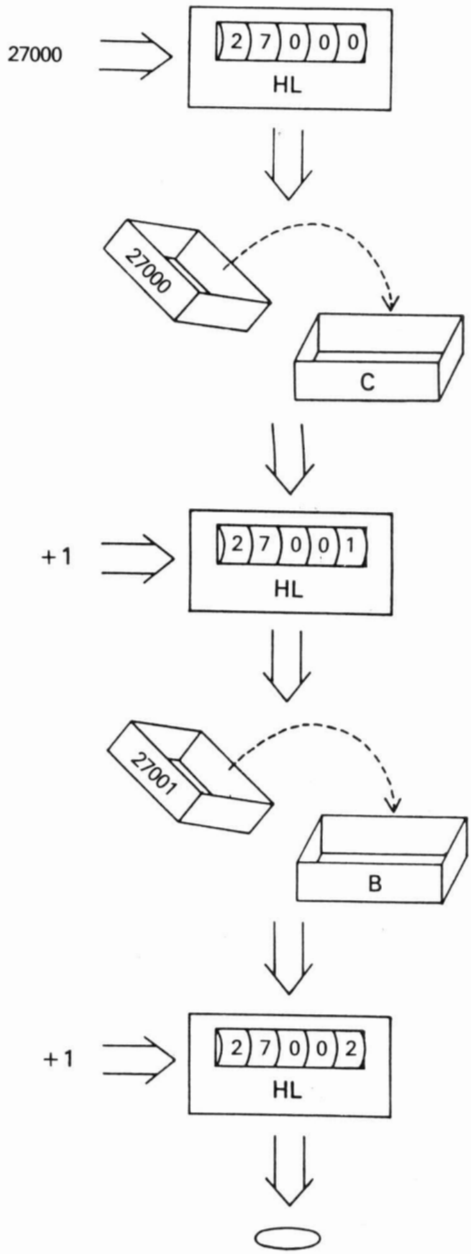


Figura 22

Ahora tenemos el mismo problema de antes: tenemos que trasladar el contenido del registro doble BC en HL. BC ha sido utilizado para almacenar temporalmente los valores mientras HL se utilizaba como un puntero. Así:

"Traslada B a HL".

"Traslada C a L".

La figura 22 nos muestra con exactitud qué es lo que pasa.

El proceso de transferencia de datos desde el BASIC al Código Máquina se ha terminado. Han sido procesados y almacenados de nuevo de forma tal que son accesibles a través del BASIC. Por consiguiente la redacción del programa está casi completa. Lo único que falta es devolver el control del Código Máquina otra vez al BASIC.

El comando USR que utilizamos para trasladar el control del BASIC al Código Máquina ya ha sido explicado y hemos visto que actúa como un comando GOSUB de BASIC. Ahora el puntero de la línea deberá ser trasladado de la línea actual a la línea asignada por la subrutina. Una vez que la subrutina llega a su fin, el resultado es el que se muestra en la figura 23.

Se coloca el comando RETURN para comunicar al ordenador que regrese a la línea inmediatamente inferior a aquella en la que se encontraba la instrucción GOSUB. Por ejemplo, si todas las líneas están numeradas en intervalos de 10, y la instrucción de GOSUB está en la línea 10, entonces la instrucción de retorno (RETURN) hará que el programa continúe trabajando desde la línea 20. En Código Máquina se utiliza la misma instrucción "RETurn" para trasladar el control desde esta instrucción de Código Máquina, a la línea inmediatamente después en la que se encuentra la instrucción USR del BASIC. Con la instrucción USR en la línea 10, y siendo la línea siguiente la número 20, una vez que la rutina en Código Máquina haya finalizado y encontrado la instrucción RET, el programa en BASIC se continuará ejecutando a partir de la línea 20.

Por lo tanto, la última instrucción de un programa en Código Máquina será "RETURN (retornar o regresar) al BASIC".

Estudio de todas las instrucciones del programa examinado

La experiencia nos dice que, antes de continuar, debemos hacer una lista clara y lógica de todas las instrucciones que componen el programa.

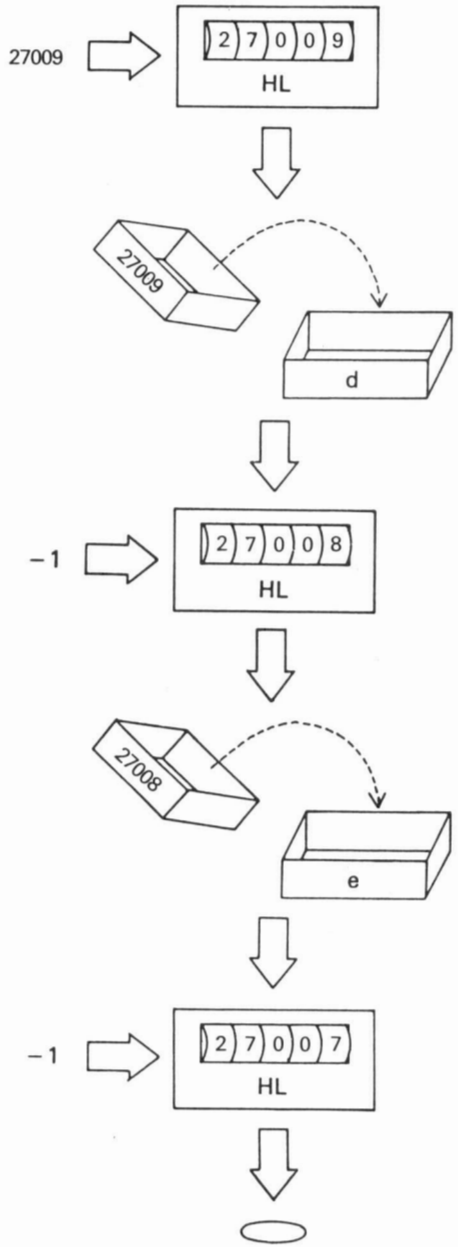


Figura 23

La primera etapa nos muestra el BASIC requerido para trasladar los valores iniciales; la segunda etapa nos muestra las distintas operaciones que realiza el programa en Código Máquina, incluyendo las transferencias de entrada y salida y las rutinas de suma y de resta. La etapa final nos muestra cómo acceder a los resultados desde el BASIC.

BASIC:

— traslada el primer número de la suma a las posiciones 27000 y 27001 (L H).
— traslada el segundo número de la suma a las posiciones 27002 y 27003 (E D).
— traslada el primer número de la resta a las posiciones 27006 y 27007 (L H).
— traslada el segundo número de la resta a las posiciones 27008 y 27009 (E D).
— salto a la subrutina de Código Máquina.

CODIGO MAQUINA:

— carga el registro doble HL con 27000.
— traslada el contenido de la posición HL a C.
— incrementa en 1 el valor de HL.
— traslada el contenido de la posición HL a B.
— incrementa en 1 el valor de HL.
— traslada el contenido de la posición HL a E.
— incrementa en 1 el valor de HL.
— traslada el contenido de la posición HL a D.
— traslada B a H.
— traslada C a L.
— traslada L a A.
— suma E con A.
— traslada A a L.
— traslada H a A.
— suma a A el registro D y el acarreo.
— traslada A a H.
— traslada el contenido de HL a las posiciones 27004 y 27005.
— no opera.
— no opera.
— carga el registro doble HL con 27009.

— traslada el contenido de la posición HL a D.
— disminuye en 1 el valor de HL.
— traslada el contenido de la posición HL a E.
— disminuye en 1 el contenido de HL.
— traslada el contenido de la posición HL a B.
— disminuye en 1 el valor de HL.
— traslada el contenido de la posición HL a C.
— traslada B a H.
— traslada C a L
— pone a cero el señalizador de acarreo.
— resta con acarreo al registro doble HL el contenido del registro doble DE.
— traslada H a B.
— traslada L a C.
— regresa al BASIC.

BASIC

— escribe el valor final del registro doble BC (resultado de la resta).
— toma el resultado de la suma por medio de PEEK y luego lo escribe mediante la sentencia PRINT.

Conversión de instrucciones al lenguaje ensamblador

Ahora ya tenemos claro qué es lo que esperamos que haga el ordenador y este es el momento más apropiado para realizar la conversión de nuestras instrucciones en mnemónicos de lenguaje ensamblador, las cuales pueden ser introducidas en un ensamblador. También sería una buena idea en este momento averiguar los códigos hexadecimales de las instrucciones, las cuales también pueden ser introducidas a través de un monitor en BASIC o a través de un programa corrector. De paso nos sirve también de comprobación de lo que hemos hecho. A continuación hay una tabla completa en la que se muestran la instrucción, el mnemónico en lenguaje ensamblador y el código hexadecimal:

Instrucciones comentadas	Lenguaje ensamblador	Códigos hexadecimales
CARGA HL CON 27000	LD HL,27000	21 78 69
PASA A C EL CONTENIDO DE LA POSICION HL	LD C,(HL)	4E
INCREMENTA HL EN 1	INC HL	23
PASA A B EL CONTENIDO DE LA POSICION HL	LD B,(HL)	46
INCREMENTA HL EN 1	INC HL	23
PASA A E EL CONTENIDO DE LA POSICION HL	LD E,(HL)	5E
INCREMENTA HL EN 1	INC HL	23
PASA A D EL CONTENIDO DE LA POSICION HL	LD D,(HL)	56
PASA B A H	LD H,B	60
PASA C A L	LD L,C	69
PASA L A A	LD A,L	7D
SUMA E A A	ADD A,E	83
PASA A A L	LD L,A	6F
PASA H A A	LD A,H	7F
SUMA A A, D Y EL ACARREO	ADC A,D	8A
PASA A A H	LD H,A	67
PASA HL A LAS POSICIONES 27004 Y 27005	LD(27004),HL	22 7C 69
NO OPERA	NOP	00
NO OPERA	NOP	00
CARGA HL CON 27009	LD HL,27009	21 81 69
PASA A D EL CONTENIDO DE LA POSICION HL	LD D,(HL)	56
DISMINUYE HL EN 1	DEC HL	2B
PASA A E EL CONTENIDO DE LA POSICION HL	LD E,(HL)	5E
DISMINUYE HL EN 1	DEC HL	2B

Instrucciones comentadas	Lenguaje ensamblador	Códigos hexadecimales
PASA A B EL CONTENIDO DE LA POSICION HL	LD B,(HL)	46
DISMINUYE HL EN 1	DEC HL	2B
PASA A C EL CONTENIDO DE LA POSICION HL	LD C,(HL)	4E
PASA B A H	LD H,B	60
PASA C A L	LD L,C	69
PONE A 0 EL SE/ALIZADOR DE ACARREO	AND 0	E6 00
RESTA CON ACARREO HL MENOS DE	SBC HL,DE	ED 52
PASA H A B	LD B,H	44
PASA L A C	LD C,L	4D
REGRESAR AL BASIC	RET	C9

Ejecución «en seco»

Ahora es el momento de comprobar cuidadosamente el programa por si acaso apareciera algún error que pudiera echar por tierra el programa. Si hay un error importante lo más probable es que el ordenador quede seriamente afectado sin darte la oportunidad de determinar dónde has fallado. La mejor forma de encontrar los «bugs» («bichos» o errores) desde luego no es hacerlo una vez que ya nos hemos molestado en teclear el programa. Este es el momento del proceso en el que se suele practicar la buena costumbre de la «ejecución en seco».

Lo que queremos decir con la «ejecución en seco» de un programa es, en efecto, la ejecución del programa ¡en el papel! Puedes, por ejemplo, utilizar una «tabla» con todos los registros, colocando sus valores después de cada instrucción, tomando nota del estado en que se encuentra el señalizador de acarreo, etc. Normalmente los errores aparecen rápidamente si utilizamos este método. Otra cosa muy ventajosa de este método de comprobación es que crea un documento que muestra qué hace el programa en cada momento, lo que nos permite realizar las consiguientes verificaciones

H	L	D	E	B	C	A	(Acarreo) si se conoce

Gráfico M

y alteraciones. Incluyo aquí una tabla sencilla que nos ayudará a realizar la «ejecución en seco» de nuestros programas. Examina cada uno de los mnemónicos del lenguaje ensamblador, observa lo que hace y luego pon en la tabla el resultado de la instrucción. Por ejemplo, si tenemos en cuenta el comando LD A, 40, entonces pon un 40 en la columna A. Si lo utilizamos de este modo nos proporcionará una «tabla preparada» (gráfico M) que nos permitirá una verificación inmediata del contenido de los registros. Si amplías la tabla de forma que comprenda todas las posiciones de memoria utilizadas por el programa, te ayudará a familiarizarte con su utilización.

Introducción del programa en el ensamblador

Aunque hayamos listado los códigos hexadecimales, no es normal trabajar con ellos si es que estamos utilizando un ensamblador ya que estaríamos haciendo el trabajo que haría dicho ensamblador. En este momento ya tenemos preparado el ensamblador y también tenemos el programa que queremos ensamblar.

Una vez que hayamos cargado (LOAD) el ensamblador se nos darán las instrucciones para manejarlo. En este caso usaremos un

comando «GO» para dejar sentado que hay un programa esperando para ser ensamblado. Luego se utiliza la instrucción «ORG» junto con la dirección de la posición de memoria, para identificar en dónde se va a guardar el programa ensamblado. Aunque ya estemos preparados para introducir los mnemónicos del lenguaje ensamblador después de las sentencias REM, vale la pena poner un comentario al inicio del programa por si fuera necesaria su identificación en un momento posterior. Esto se hace poniendo un signo de admiración (!) después de la sentencia REM, seguido del comentario (*). En este caso sería apropiado escribir «rutina de suma y resta».

Debemos introducir cada mnemónico cuidadosamente asegurándonos de no cometer ningún error. Comprueba y contrasta la lista final de mnemónicos con la lista original o inicial. Es posible poner más de una instrucción en cada línea, pero separándolas por punto y coma (;), pero resulta mucho más fácil leer la rutina si sólo utilizamos una instrucción por línea. Así debe aparecer la lista de instrucciones una vez que ha sido introducida en el ensamblador:

```
ORG 28000
LD HL,27000        217869
LD C,(HL)          4E
INC HL             23
LD B,(HL)          46
INC HL             23
LD E,(HL)          5E
INC HL             23
LD D,(HL)          56
LD H,B             60
LD L,C             69
LD A,L             7D
ADD A,E            83
LD L,A             6F
LD A,H             7C
ADC A,D            8A
LD H,A             67
```

* *N. del T.:* Recordemos que el autor se está refiriendo al programa ensamblador analizado en el capítulo anterior. Se trata del ensamblador «Ultra Violet» de ACS Software. No obstante, se puede conseguir la ejecución de este programa a través de los programas monitor (Capítulo 6) o el cargador hexadecimal (Capítulo 1) que se incluyen en el libro.

```
LD (27004),HL      227C69
NOP                00
NOP                00
LD HL,27009        218169
LD D,(HL)          56
DEC HL             2B
LD E,(HL)          5E
DEC HL             2B
LD B,(HL)          46
DEC HL             2B
LD C,(HL)          4E
LD H,B             60
LD L,C             69
AND 0              E600
SBC HL,DE          ED52
LD B,H             44
LD C,L             4D
RET                C9
```

Final del programa

Ya estamos casi a punto de ensamblar la rutina pero todavía quedan algunas cosas por hacer.

Pon un comentario en la línea siguiente a la última para indicar que es el final de la rutina. Las instrucciones «Go», «Finish» y «ORG» son muy importantes para el propio ensamblador por lo que rechazará cualquier operación si son omitidas. El ensamblaje se detendrá y aparecerá un código de error. Si se ponen demasiados espacios o éstos son insuficientes entonces puede ser mal interpretado por el ensamblador. Esto puede provocar graves problemas ya que es un error difícil de detectar.

Una vez que el programa ha sido ensamblado es recomendable hacer una copia en cinta cassette o en microdrive (disco magnético) de forma que si el programa «casca» no se haya perdido todo. Para grabar (SAVE) el programa ensamblado teclea:

SAVE «nombre» CODE xxxxx, yyyyy

en donde xxxxx representa la posición en la que quieres colocar la rutina que vas a grabar e yyyyy es el número de bytes de dicha

rutina. La parte fundamental la forma la instrucción CODE, que es la que comunica al ordenador que vas a grabar Código Máquina. Para verificar la grabación basta con teclear VERIFY "" CODE. Para grabar el programa BASIC que contiene los mnemónicos resulta más fácil cambiar de sistema. Intenta hacerlo como si se tratara de un programa normal. Es decir: SAVE «nombre».

Para cargar el Código Máquina en su forma hexadecimal, escribe LOAD «nombre» CODE. Esto cargará en Código Máquina el programa llamado «nombre», y lo pondrá en las direcciones de donde ha sido grabado. Para recargar un programa BASIC normal, es decir, el programa que contiene los mnemónicos del lenguaje máquina, escribe LOAD «nombre».

El programa en BASIC utilizado para guardar los mnemónicos antes de comenzar con el ensamblaje ha perdido su utilidad por lo que debe ser borrado. No trates de hacer esto a través del comando NEW —a no ser que quieras perderlo todo—. Utiliza CLEAR xxxxx (en donde xxxxx es la posición hasta donde quieras que llegue la «protección», menos uno). De esta forma se establece un nuevo RAMTOP y ahora ya podemos hacer NEW. Puedes estar seguro de que el Código Máquina ahora está sano y salvo, y que en la memoria sólo queda el ensamblador.

Ahora vamos a ver el programa de control que organiza el traslado de las variables a la memoria, las vuelve a sacar y escribe los resultados en la pantalla.

```
 10 POKE 27000.4: POKE 27001,1:
POKE 27002,3: POKE 27003,2
 20 POKE 27006,7: POKE 27007,1:
POKE 27008,8: POKE 27009,9
 30 PRINT "RESTA = ";USR 28000
 40 PRINT "SUMA = ";PEEK 27004+
(PEEK 27005)*256
 50 STOP
```

Sería conveniente poder cargar el programa de control (escrito en BASIC) casi al mismo tiempo que el programa en Código Máquina. Esto puede lograrse mediante la autoejecución del programa BASIC, cuya primera línea sería la encargada de cargar el programa

en Código Máquina. Ambos pueden grabarse en una cinta, uno al lado del otro.

Una vez verificada la copia del programa sólo nos falta su ejecución. Esto se realiza mediante las instrucciones GOTO o RUN del programa BASIC ya que la instrucción PRINT USR 28000 ya está comprendida en el programa BASIC. Puedes experimentar con valores diferentes dentro de los registros y observa (¿predice?) los resultados.

Saltar es divertido

Capítulo 8

SALTANDO DE UN LUGAR A OTRO

Hasta ahora nos hemos dedicado, como ocupación principal, al desarrollo de una «buena técnica». Ahora es el momento de ampliar el repertorio de comandos disponibles para ser utilizados en la programación. En realidad son necesarios si has dado rienda suelta a tus instintos creadores. Cuanto más sepamos, más posibilidades tendremos. Cada vez se pondrá más de manifiesto el hecho de que de unos cuantos comandos podremos obtener muchas combinaciones distintas, lo cual proporciona una gran versatilidad. Este capítulo tratará acerca de saltos sencillos, saltos condicionales, saltos relativos, saltos condicionales relativos, llamadas y llamadas condicionales. Además se tendrá en cuenta el registro PC, la instrucción RET y otros aspectos relacionados con el funcionamiento interior del ordenador.

El registro PC

En términos generales ya hemos hablado acerca de la organización de los registros de la CPU y también hemos prestado mayor atención a aquellos que son más fáciles de utilizar (HL, BC, etc.,). Pero hay otros registros dobles a los que, o bien no tenemos acceso directo, o bien sólo podemos acceder a ellos por medio de operaciones relativamente fáciles. Esto es así debido a la necesidad de la

CPU de reservar espacio para almacenar sus propias variables en un sitio en el que se encuentran tan accesibles para ella como protegidas de cualquier interferencia exterior. El registro PC es un buen ejemplo de ello. Este registro doble almacena la dirección de la posición de memoria del comando que está siendo ejecutado y, por consiguiente, funciona como un puntero de la CPU dándole la posición de comienzo de la instrucción siguiente.

Cuando conectamos el ordenador, el valor del registro PC es 0000, por lo que el primer comando a ejecutar por el ordenador será el que esté situado en la primera posición de la memoria. Esta es la razón por la que la ROM del Sinclair está mapeada (colocada) en los primeros 16K de la memoria. Dicho con otras palabras, por medio de la ROM, los valores de las posiciones de 0 hasta la 16383 están predeterminadas y, por este motivo, cada vez que se conecta el ordenador, va inmediatamente a lo que es llamado como la «rutina inicial». Una vez dentro de una rutina en Código Máquina, el registro PC siempre contiene la posición de la instrucción en curso, de forma que, cuando se haya ejecutado dicha instrucción, el registro PC cambia de acuerdo con la longitud de la instrucción. Por lo tanto, ya conoce la posición de la instrucción siguiente. Podemos presumir que si cambiáramos desde fuera el valor interno del registro PC, sería posible cambiar la siguiente posición a la que se dirige la CPU.

La instrucción de salto

El primer problema que hay que resolver es que no hay comandos que alteren directamente el contenido del registro PC del mismo modo que, por ejemplo, los hay para los registros HL o BC. Afortunadamente hay una instrucción que cambia el valor interno del registro PC llamada la instrucción JUMP (SALTO) o JP. Funciona de forma parecida a la instrucción GOTO del BASIC. Basta con cargar el registro PC con el valor requerido, que será la posición en la que se encuentra la instrucción que queremos que se ejecute a continuación. En BASIC, si estamos en la línea 100, y queremos que la siguiente instrucción que se ejecute sea la de la línea 1050, entonces tendremos que utilizar la instrucción GOTO 1050 en dicha

línea 100. En Código Máquina la operación es exactamente la misma. Si resulta que cuando tenemos una pequeña rutina en las posiciones 30000 hasta la 30012 y, una vez que sea totalmente ejecutada, deseamos continuar con la ejecución de una rutina que se encuentra en las posiciones 32000 en adelante, el proceso tendrá este aspecto:

30000 — 30012	:	primera rutina
30013 JP 32000	:	salto a la posición 32000
32000 —	:	segunda rutina

A continuación se muestra un ejemplo de esta instrucción:

```
ORG 30000
30000 3E 00          LD A,0
30002 06 07          LD B,7
30004 C3 00 7D       JP 32000
32007 60             LD H,B
32008 6F             LD L,A
32009 C9             RET
```

Así como, sin duda, resulta muy útil poder saltar de una zona de la memoria a otra y ejecutar las instrucciones de esa zona de memoria, ¿en cuántos casos es necesario escribir una rutina en dos zonas diferentes de la memoria? ¿No sería más útil poder ejecutar un comando de otro lugar de la memoria sólo si una determinada situación es verdadera o falsa? Por ejemplo, si en un programa BASIC que realice un cálculo muy complicado, se le pidiera al ordenador que indicara cuándo el resultado sea más alto o más bajo que el esperado, se utilizaría una rutina parecida a ésta:

500 IF (si) el resultado es mayor que el esperado THEN (entonces) GOTO (ir a) 1000
510 IF el resultado es menor o igual que el esperado THEN GOTO 1500
1000 PRINT (escribe) "mayor que"
1010 STOP (paro)
1500 PRINT "menor o igual que"
1510 STOP (paro)

Fíjate que en este programa en BASIC se ha utilizado la sentencia STOP para finalizar la rutina y que no existe una instrucción paralela en Código Máquina. Bueno, no del todo ya que nunca se necesita una parada total. Al terminar una rutina en Código Máquina siempre es necesario regresar al BASIC. La función normal de la instrucción RET es la de hacer regresar al ordenador al control del BASIC.

Las instrucciones de salto condicional más utilizadas

A continuación se muestra una lista de las instrucciones de salto condicional más utilizadas:

JP z	:	salta si cero.
JP nz	:	salta si no cero.
JP c	:	salta si el señalizador de acarreo está marcado con 1.
JP nc	:	salta si el señalizador de acarreo está puesto a cero.
JP m	:	salta si es negativo.
JP p	:	salta si es positivo.

JP z quiere decir que, en caso de que el «señalizador de cero» (¡ojo!, se trata de un nuevo señalizador) valga uno, entonces se ejecutará la instrucción que se encuentre determinada después de la del salto. Por lo tanto, el resultado del cálculo anterior era cero. Si el resultado fuera distinto de cero, y esta condición fuera falsa, entonces se ejecutaría la instrucción situada en la posición siguiente de memoria.

Por ejemplo:

```
ORG 28000
28000 3E 00        LD A,0
28002 FD 69        ADD 0
28004 CA 6B 6D     JP Z,28011
28007 3D           DEC A
28008 C3 62 6D     JP 28002
28011 C9           RET
```

NOTA: En este caso concreto el acumulador ya contiene un 0, por lo que si le sumamos un 0 el resultado final también sería 0, dando lugar al cumplimiento de la condición. Esto provocaría el salto a la dirección 28011 en donde se regresaría al BASIC.

En este programa la prueba consiste en determinar si el resultado de sumar un cero al acumulador es o no es igual a cero. Si es igual a cero, entonces el control del programa volverá al BASIC. Si no lo es, entonces el valor del acumulador será disminuido; es decir, sería igual a sí mismo menos 1, y luego probaría de nuevo si sumando un cero al mismo el resultado sería cero, y continuaría haciendo esto hasta que el acumulador fuera igual a cero. Asegúrate de que la instrucción utilizada antes del salto condicional modifica de hecho este señalizador. En caso de duda consulta la tabla que se encuentra al final del libro.

JP nz quiere decir que se saltará a la dirección fijada sólo en caso de que el señalizador de cero esté puesto a cero, por lo que el resultado de la última operación no era igual a cero. Si, como en el ejemplo anterior, la instrucción precedente fuera «sumar cero», seguida de JP nz, 32000, entonces el Código Máquina tendría este aspecto:

```
29999 3D            DEC A
30000 FD 69         ADD 0
30002 CA «inicio»   JP nz, 29999
30005 C3 «32000»    JP 32000
```

El equivalente en BASIC sería:

IF A es mayor o igual a 0 THEN GOTO 32000

JP c, quiere decir «salto sólo si el señalizador de acarreo está activado o marcado con un 1». Dicho de otra forma, en caso de que el último cálculo haya provocado un desbordamiento, entonces se activará con un 1 el señalizador de acarreo y se cumplirá, por tanto, la condición. Un ejemplo sencillo consistiría en hacer que, cuando los registros A y B, en combinación, sean mayores que 255, se salte

a una nueva dirección, pero si no es así, que continúe. A continuación se ofrece un programa que realiza lo que acabamos de decir:

```
ORG 30000
30000 80          ADD A,B
30001 DA 30 75    JP C,30000
```

El equivalente en BASIC sería: IF A más B es mayor que 255 THEN GOTO 30000. Esto nos proporciona un sistema de comprobación de posibles errores cuando la suma de dos variables es un número demasiado grande para ser manejado por los registros de la CPU.

La instrucción JP nc es muy parecida a la anterior excepto que el salto sólo se producirá si el señalizador de acarreo está puesto a cero. Esto comprueba si el resultado del último cálculo ha provocado o no un desbordamiento. En caso de que no se haya producido —los valores sumados son menores que 256— entonces saltará a la posición fijada. A continuación vemos un ejemplo:

```
ORG 30000
30000 80          ADD A,B
30001 D2 30 75    JP NC,30000
```

El equivalente en BASIC será:

```
IF A+B<256 THEN GO TO 30000
```

Las dos últimas condiciones disponibles para realizar un salto son las de número negativo o positivo. Se cumplirán según que el señalizador del signo (otro nuevo señalizador) refleje un valor positivo o negativo. Como ya hemos visto de cerca este señalizador basta asegurar que JP p quiere decir «salto si el señalizador del signo refleja un número positivo». El comando JP m quiere decir «salto si el señalizador del signo refleja un número negativo».

A lo largo de estos ejemplos hemos utilizado las direcciones para mostrar con exactitud qué era lo que sucedía pero si empleamos un ensamblador, o si escribimos un programa que pueda ser colocado en cualquier lugar de la memoria, es mucho más fácil utilizar «etiquetas». Para ir a una rutina y luego regresar otra vez al principio,

etiquetá el comienzo del programa con INICIO, y al final ejecuta la instrucción JP INICIO. Cuando utilizamos un ensamblador no tenemos la necesidad de emplear direcciones ya que el ensamblador lo hará por ti, averiguando las direcciones de cada parte del programa. También es posible etiquetar ciertos bytes dentro de un programa y pueden utilizarse para almacenar resultados o cálculos realizados por una rutina. Se trata de una de las funciones más útiles, aparte, por supuesto, de la conversión de los propios mnemónicos. A continuación se muestra un breve ejemplo de cómo pueden utilizarse las etiquetas en un programa. No te preocupes si aparece un tanto obscura la forma en que funciona el programa. Limítate a tomar nota de las etiquetas y de cómo son utilizadas y asignadas.

1. Listado BASIC del ensamblador ACS.

```
  1 REM GO
  2 REM ORG 30000
 10 REM LD HL,22527
 20 REM LD B,192
 30 REM !PARA DIFERENTES BLOQUE
S DE PANTALLA CAMBIAR "B" Y "HL"
 40 REM A;LD C,32
 50 REM B;LD E,(HL)
 60 REM RL (HL)
 70 REM DEC HL;DEC C
 80 REM JR NZ,B
 90 REM BIT 7,E
100 REM JR Z,C
110 REM PUSH HL;LD DE,32
120 REM ADD HL,DE
130 REM SET 0,(HL)
140 REM !PONER UN "0" EN (HL) P
ARA EVITAR BUCLE SIN FIN
150 REM POP HL
160 REM C;AND A
170 REM DJNZ,A
180 REM RET
190 REM FIN
```

2. Listado ensamblado.

```
ORG 30000
30000 21 FF 57        LD HL,22527
30003 06 C0           LD B,192
CAMBIAR 'B' Y 'HL' PARA DIFEREN-
TES BLOQUES DE PANTALLA
A
30005 0E 20           LD C,32
B
30007 5E              LD E,(HL)
30008 CB 16           RL (HL)
30010 2B              DEC HL
30011 0D              DEC C
30012 20 F9           JR NZ,B
30014 CB 7B           BIT 7,E
30016 28 08           JR Z,C
30018 E5              PUSH HL
30019 11 20 00        LD DE,32
30022 19              ADD HL,DE
30023 CB C6           SET 0,(HL)
RES 0,(HL)PARA EVITAR BUCLE SIN
FIN
30025 E1              POP HL

C
30026 A7              AND A
30027 10 E8           DJNZ,A
30029 C9              RET
```

Desde luego, es posible saltar a una posición del programa colocada en un sitio anterior. Digamos que se trata de la posición 30000, y necesitamos ejecutar la rutina situada en 28000. Es perfectamente posible utilizar la instrucción JP 28000 en la posición 30000.

Hay un factor muy importante al que todavía no nos hemos enfrentado. Se trata de que, cuando saltamos hacia atrás formando un bucle que se llama a sí mismo, resulta que no existen instrucciones de RETorno entre las mismas. Esto es la forma más rápida de hacer que el ordenador entre en un bucle sin fin —¡sin esperanzas de escaparse de él!—. Para salirse de un bucle de este tipo en BASIC sólo hace falta pulsar la tecla BREAK, pero en Código Máquina no tenemos esta posibilidad. Algunos ordenadores tienen la posibilidad

de autocontrol de forma que se puede empezar de nuevo sin perder las rutinas que se encuentran en la memoria pero esto no es posible con el ZX80, ZX81 o el ZX Spectrum. La única forma de salir de un bucle sin fin es «desconectando el enchufe». Esto es así porque el Código Máquina del Spectrum no acusará la pulsación de la tecla BREAK o cualquier otra tecla con ese propósito, a no ser, por supuesto, que haya incluida en el programa una rutina que explore el teclado.

Acabamos de ver los saltos directos que nos permiten cargar el registro PC, o el Contador del Programa, con una nueva dirección con el fin de alterar la posición de la que será tomada la instrucción siguiente. Pero hay otro tipo de instrucción de salto muy útil cuando escribimos pequeñas rutinas que no dependen de la zona de memoria en la que se encuentran.

El salto relativo

Este tipo de salto es conocido como el SALTO RELATIVO. La razón por la que se le da esta denominación es que nos permite saltar a una posición «más o menos un número relativo con respecto a la posición en curso». En lugar de saltar, digamos, a la posición 28000 cuando nos encontramos en la posición 28005, se puede utilizar un salto relativo de «menos 5». Esto puede parecer una forma un tanto complicada de saltar de una posición a otra pero tiene la ventaja de que podemos colocar la rutina en cualquier parte de la memoria. Al utilizar los saltos relativos, debido a que no hay que señalar ninguna dirección en la instrucción de SALTO, podemos desplazar la rutina de 28000 a 24000 o cualquier otra posición de la memoria sin necesidad de realizar una nueva conversión.

Esta posibilidad es particularmente útil cuando una rutina puede ser utilizada más de una vez dentro de un programa muy extenso o cuando cambiamos el programa para ser utilizado por un ordenador de mayor tamaño. Del mismo modo puede mostrarnos sus ventajas cuando no hay sitio suficiente en la memoria para introducir una rutina, y es necesario desplazarla hacia abajo para dejar sitio. El comando del salto relativo puede ahorrarnos gran cantidad de trabajo más adelante. Este comando tiene una importante limitación: sólo

es posible saltar a una distancia máxima de «más 129 bytes» o «menos 126 bytes» con relación a la dirección actual o en curso. Esto puede parecer demasiado limitado pero, en la práctica, «más 129 bytes y menos 126 bytes» suelen proporcionar un espacio lo suficientemente amplio como para que quepa una rutina de tamaño estándar. Es complicado calcular desplazamientos mayores que éstos. Por consiguiente, basta decir que la instrucción de salto relativo (JR) podemos tomarla como una instrucción local.

El cálculo del desplazamiento de un salto relativo resulta bastante fácil, aunque requiere poner cierta atención. No es posible utilizar directamente números negativos dentro de un programa en Código Máquina. Para realizar un desplazamiento con un salto relativo que es positivo, es decir, entre 0 y 129, el número que debemos introducir en el ordenador será igual a dicho desplazamiento. Cuando el desplazamiento sea negativo, es decir, entre 0 y menos 126, entonces el número que debemos introducir en el ordenador será 256 más el desplazamiento negativo. Esto es posible hacerlo, si utilizas el ensamblador empleado en este libro, simplemente tecleando el desplazamiento, positivo o negativo y, de esta forma, no es necesario que te preocupes de calcular dicho desplazamiento.

Cuando se ejecuta una instrucción de salto relativo, el ordenador está preparado para ir a una instrucción que se encuentra un cierto número de bytes más arriba (o más abajo) de la posición en la que está en ese momento. Dicho con otras palabras, si el ordenador ejecutara la instrucción JR 0 (salto relativo de 0 bytes), entonces esto no funcionaría ya que, en ningún caso, el ordenador podría llegar a ejecutar el comando de la dirección siguiente. Lo único que haría sería incrementar en 0 el valor de PC. De todas formas, debido a que la longitud de la instrucción de salto relativo es de dos bytes, si ejecutáramos la instrucción JR −2 (salto relativo de «menos 2 bytes») se volvería a ejecutar la instrucción JR −2, lo cual sería un bucle sin fin. La única manera de salirse de él sería tirando del enchufe y desconectando el ordenador.

A continuación hay un ejemplo de la utilización del comando de salto relativo y también se puede ver lo útil que es cuando se combina con el uso de etiquetas en el ensamblador.

```
ORG 30000
START
30000 3E 29          LD A,41
30002 06 28          LD B,40
30004 18 06          JR PART2
30006 00             NOP
30007 00             NOP
30008 00             NOP
30009 00             NOP
30010 00             NOP
30011 00             NOP
PART2
30012 0E 0E          LD C,14
30014 FD 69          ADD C
30016 18 02          JR +2
30018 18 EC          JR START
30020 C9             RET
```

Los saltos relativos condicionales

Es interesante advertir que en ningún momento del programa se utilizan direcciones. Esta rutina puede ser colocada en cualquier lugar de la RAM del ordenador sin sufrir ninguna modificación.

También es posible realizar saltos relativos condicionales exactamente de la misma forma que los saltos condicionales. La única diferencia es que hay menos tipos distintos de saltos relativos condicionales disponibles y sólo es posible hacer los siguientes:

JR Z : salto relativo si está marcado con un 1 el señalizador de cero.
JR NZ : salto relativo si el señalizador de cero está puesto a 0.
JR C : salto relativo si el señalizador de acarreo está marcado con un 1.
JR NC : salto relativo si el señalizador de acarreo está puesto a 0.

Se utilizan exactamente de la misma forma que las versiones correspondientes de las instrucciones de salto, acordándonos de utilizar el desplazamiento en lugar de una dirección.

En el BASIC, así como disponemos de la instrucción GOTO, también tenemos la instrucción GOSUB. Esta nos permite ejecutar una rutina diferente al programa principal y luego regresar al programa principal sin necesidad de otra instrucción GOTO. Por consiguiente, resulta más fácil la realización de pequeñas subrutinas. Una gran ventaja es la posibilidad de reutilizar una rutina determinada muchas veces dentro del mismo programa. Por ejemplo, para llevar a cabo una operación concreta que requiera un cierto número de comandos, tres o cuatro veces dentro del msimo programa, no será necesario ponerla más de una vez en el programa principal sino que haremos que dicha operación actúe como una subrutina. Da lo mismo colocar la subrutina antes o después del programa principal, y se puede acceder a ella a través del comando GOSUB del BASIC. Una vez que se haya ejecutado completamente, la instrucción RETURN, colocada al final de la subrutina, hará que el control del ordenador vuelva a la sentencia inmediatamente siguiente a aquella en la que le encontraba el GOSUB del programa principal. Para llamar a una subrutina en Código de Máquina desde un programa en Código Máquina se necesita la instrucción «CALL» (LLAMADA), seguida de la dirección (o la etiqueta si es que estás utilizando un ensamblador) de la subrutina.

He aquí un ejemplo de cómo se puede hacer esto en un programa:

```
10 RANDOMIZE USR 30000
ORG 30000
START
30000 CD 6E 6D        CALL,28014
30003 C9              RET


ORG 28014
SUBRT
28014 3E 00           LD A,0
28016 06 00           LD B,0
28018 C9              RET

20 PRINT "FIN DE LA RUTINA"
```

Otra utilización de la instrucción CALL es la de ejecutar subrutinas que ya se encuentran en la ROM. Aunque muchas de las rutinas de la ROM están muy entremezcladas con la ejecución del BASIC, y otras pertenecen a la propia ROM, todavía hay muchas de ellas que pueden sernos útiles. Algunas de estas rutinas son muy cortas si es que la única razón para utilizar la instrucción CALL es la de ahorrar tiempo al copiarlas en la RAM y en nuestros programas. De todas formas, muchas de ellas son largas y complicadas, lo que hace que consumamos mucho tiempo al copiarlas y además, en lo que a memoria se refiere, produce un gran despilfarro de espacio. Además la utilización de la instrucción CALL elimina en este caso toda posibilidad de error.

Una vez que una subrutina ha sido ejecutada es necesario regresar al programa principal. En BASIC esto se consigue utilizando la instrucción RETURN: en Código Máquina, se hace a través de la instrucción RET. Esto puede parecer un tanto confuso ya que, hasta ahora, la instrucción RET ha sido utilizada para hacer que el control del ordenador vuelva al BASIC cuando se coloca al final de una rutina en Código Máquina. Lo cierto es que la instrucción RET realiza un trabajo muy sencillo: devolver el control a la posición en donde se ha utilizado la instrucción CALL o GOSUB. Por lo tanto, cuando una rutina en Código Máquina sea ejecutada mediante la instrucción del BASIC USR, y se llegue a una instrucción RET, entonces el control regresará al BASIC. Del mismo modo, una vez ejecutada una instrucción CALL en Código Máquina, si al final de la subrutina quieres volver al programa principal, la instrucción RET permitirá que esto suceda.

Una vez que se ejecuta una llamada (CALL), se almacena la posición en la que se encuentra de forma que, cuando se llega a una instrucción RET, el ordenador es capaz de regresar al lugar correcto de la memoria. Cuando es encontrada una instrucción RET y el control ya se ha devuelto al programa principal, el valor de la posición almacenado por el ordenador es sustituido por el valor siguiente. Dicho de otra forma, cuando accedemos a un programa en Código Máquina a través de la instrucción BASIC USR, la dirección guardada por el ordenador es la de la siguiente instrucción BASIC. Cuando accedemos a una subrutina en Código Máquina desde el Código Máquina, a través de la instrucción CALL, en la CPU también se alma-

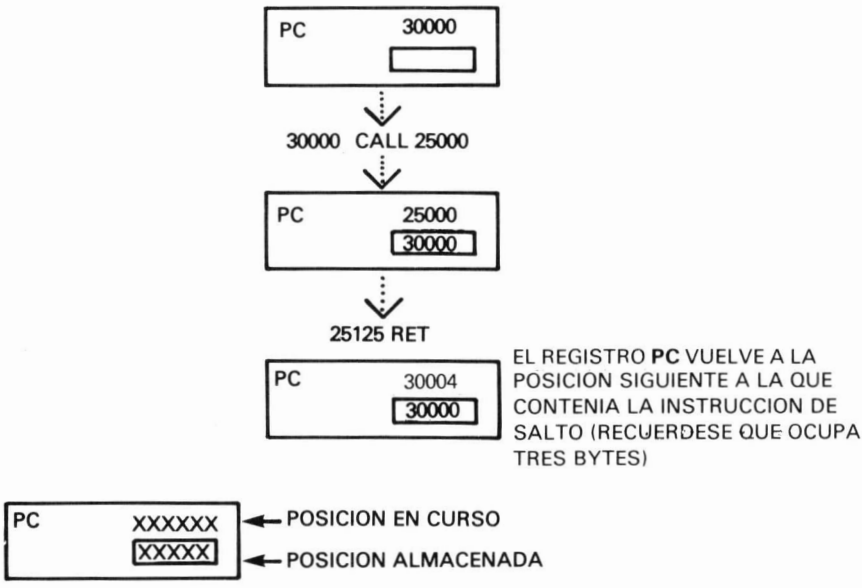


Figura 24

cena la dirección de dicha instrucción. Si mientras se ejecuta la subrutina aparece una instrucción RET, regresará a la dirección almacenada por el ordenador. En este caso se trataría del programa principal de Código Máquina. Una vez finalizada dicha subrutina, el valor guardado por el ordenador será aquel que estaba almacenado antes de que se llamara a la subrutina. Esta sería la dirección de la instrucción posterior en BASIC. La figura 24 muestra todo esto más claramente.

También es posible realizar instrucciones de llamadas condicionales del mismo modo que en los saltos y en los saltos relativos. De hecho funcionan exactamente igual que las que acabamos de explicar por lo que no pretendo insistir más en ello. Una vez más la utilización de etiquetas con las instrucciones CALL incrementa de forma espectacular la facilidad con que pueden utilizarse en un programa y es igualmente extraordinaria la aportación que constituyen para la creación de programas estructurados. Es posible dividir un programa en series de subrutinas a las que luego podríamos acce-

der por medio de un breve programa maestro. Trabajando de esta forma se puede lograr que el listado sea más comprensible una vez que hayamos terminado y también permite que sea más fácil ajustar o ampliar el programa. Lamentablemente no hay instrucciones tales como llamadas relativas. Debido a esto hay que utilizar el diccionario directo, impidiendo que el resultado sea intercambiable. A pesar de todo, cambiar un programa de sitio no es demasiado difícil ya que, como recordarás, no es necesario acordarse de la dirección a la que habrá de regresar una vez que haya finalizado una subrutina. Se puede situar la subrutina en la memoria, y luego hacer que el programa maestro se sitúe a sí mismo. Los mnemónicos y los códigos en decimal y en hexadecimal de estas instrucciones aparecen listados en el apéndice B.

Cómo continuar con el ZX Spectrum

Capítulo 9

UTILIZACION DE LA PANTALLA Y DEL TECLADO

En la primera parte de este capítulo vamos a ver cómo se interrelacionan o «comunican» el ordenador y el usuario. Por consiguiente, el tema central lo formarán el «teclado» y la «pantalla». Empezaremos rápidamente echando una ojeada a la disposición del teclado y a la forma en que podemos utilizar las variables del sistema como método sencillo para leer el teclado. Luego veremos algunos casos prácticos y algunas rutinas que podrás incluir en tus programas. Veremos el modo en que está organizada la pantalla y cómo está mapeada la memoria. Debido a que la pantalla está mapeada, en su mayor parte, dentro de la RAM del ordenador, nos es posible cambiar con facilidad el contenido de la pantalla, aunque Clive Sinclair no nos lo ha puesto demasiado fácil a los programadores porque ha distribuido la pantalla de una forma muy especial.

Debemos prestar atención a la forma en que pueden resolverse los problemas inherentes al Spectrum, cómo realizar tareas tales como escribir los caracteres Sinclair en la pantalla y cómo definir nuevos caracteres para luego presentarlos en la pantalla. Al final habrá una rutina en Código Máquina que nos permitirá colocar (PLOT) cualquier punto de la pantalla y, debido a su asombrosa rapidez, esta rutina demostrará ser más útil de lo que parece en un principio. El teclado está dividido en filas y en columnas, lo que per-

mite al ordenador identificar, con unas pocas instrucciones, la pulsación de una tecla cualquiera. Esto también permite al ordenador registrar cuándo se han pulsado dos teclas al mismo tiempo, dando por supuesto que se encuentran en filas separadas. Ejemplos de esto son las teclas de CAPS SHIFT y SIMBOL SHIFT. Si estuvieran una al lado de la otra y en la misma fila, entonces confundirían al ordenador el cual no sería capaz de distinguir entre éstas y otras teclas pulsadas al mismo tiempo.

La lectura del teclado por el ordenador

La forma en que el ordenador lee el teclado es un tanto complicada. Nos será útil e interesante entender los conceptos fundamentales de la distribución del teclado. Afortunadamente para nosotros no tenemos necesidad de entender ni de utilizar la rutina de la ROM que explora el teclado ya que hay dos variables muy útiles que el ordenador guarda constantemente en la zona de las variables del sistema. Se encuentran entre las posiciones 23552 y 23665. En esta zona se guardan distintas variables, incluso los colores en curso, la longitud de los sonidos (BEEP) y, lo que es más importante para nosotros, la última tecla que ha sido pulsada. Haciendo PEEK en la posición 23557 resulta posible averiguar si se ha pulsado o no una tecla. Para demostrarlo, escribe el pequeño programa BASIC que se presenta a continuación:

```
  10 IF PEEK 23557=5 THEN  PRINT
 AT 0,0;"TECLA PULSADA"
  20 IF PEEK 23557<>5 THEN  PRIN
T AT 0,0;"                  "
  30 GO TO 10
```

Este programa comprueba si hay o no hay un 5 en la posición 23557 y, en caso afirmativo, escribe en la pantalla «Tecla pulsada». Si dejas de presionar la tecla, de forma que el contenido de la posición 23557 deje de ser igual a 5, borrará las palabras «Tecla pulsada» con espacios. La sentencia GOTO 10 de la línea 30 quiere decir que la rutina continúa sin fin.

CODE	CHR$	CODE	CHR$
32	■	33	■
34	■	35	■
36	■	37	■
38	■	39	■
40	■	41	■
42	■	43	■
44	■	45	■
46	■	47	■
48	■	49	■
50	■	51	■
52	■	53	■
54	■	55	■
56	■	57	■
58	■	59	■
60	■	61	■
62	■	63	■
64	■	65	■
66	■	67	■
68	■	69	■
70	■	71	■
72	■	73	■
74	■	75	■
76	■	77	■
78	■	79	■
80	■	81	■
82	■	83	■
84	■	85	■
86	■	87	■
88	■	89	■
90	■	91	■
92	■	93	■
94	■	95	■
96	■	97	■
98	■	99	■
100	■	101	■
102	■	103	■
104	■	105	■
106	■	107	■
108	■	109	■
110	■	111	■
112	■	113	■
114	■	115	■
116	■	117	■
118	■	119	■
120	■	121	■
122	■	123	■
124	■	125	■
126	■	127	■
128	■	129	■
130	■	131	■
132	■	133	■
134	■	135	■
136	■	137	■
138	■	139	■
140	■	141	■
142	■	143	■
144	■	145	■
146	■	147	■
148	■	149	■
150	■	151	■

CODE		CHR$	CODE		CHR$
152	=	I	153	=	J
154	=	K	155	=	L
156	=	M	157	=	N
158	=	O	159	=	P
160	=	Q	161	=	R
162	=	S	163	=	T
164	=	U			

Esta información nos es muy útil pero no nos dice la tecla que ha sido pulsada, cosa que se consigue haciendo PEEK en la posición 23560. A menudo a esta posición se la conoce como «Last K» («Ultima Tecla»). La letra «K» es la primera de Key = tecla), ya que guarda el valor de la última tecla pulsada y, además, continúa guardando ese número incluso después de que se haya soltado la tecla. Utiliza esto en combinación con el contenido de la posición 23557 para averiguar, en primera instancia, si se ha pulsado una tecla, y luego saber qué tecla ha sido pulsada, siempre que se haya pulsado una tecla en primer lugar. A continuación incluimos un pequeño ejemplo de un programa escrito en BASIC que nos muestra lo que pasa si sólo utilizamos el contenido de la posición 23560, es decir, el valor de la última tecla que ha sido pulsada:

```
10 PRINT PEEK 23560,: GO TO 10
```

Pruébalo y descubrirás que no tiene demasiada utilidad. Después prueba el programa que se da a continuación, el cual utiliza ambos valores combinándolos mutuamente.

```
 10 IF PEEK 23557=5 THEN  PRINT
AT 0,0;CHR$ (PEEK 23560)
 20 GO TO 10
```

Fíjate que para escribir el carácter que está siendo pulsado en ese momento es necesario utilizar la instrucción CHR$ con el contenido de la posición. El ordenador sólo almacena el número del CODE (código) de la tecla en la posición 23560. El ordenador sólo utiliza un byte para guardar el código del carácter y, por lo tanto, habrá un número máximo de 256 caracteres diferentes disponibles.

La forma efectiva de utilizar el salto relativo condicional

Esta información ya puede ser utilizada para escribir una rutina de Código Máquina que funciona de forma parecida a la de BASIC que veíamos anteriormente pero que devuelve el valor de la última tecla pulsada al registro C. La única diferencia es que sólo regresa en el caso de que se haya pulsado una tecla. Ahora ya podemos ver una manera efectiva de utilizar la instrucción de salto relativo condicional. Pero debemos tener cuidado ya que, si estamos en Código Máquina —debido a que la lectura de cada tecla se hace separadamente— el ordenador no registrará la tecla BREAK. Si no se pulsa ninguna tecla, permanecerá en Código Máquina. Esta situación puede evitarse si sabemos que va a ser pulsada una tecla y que esa tecla nos llevará directamente al BASIC una vez que se pulse, en cuyo caso ya podríamos operar con la tecla BREAK del BASIC. De todas formas, si queremos utilizar esto en un programa escrito totalmente en Código Máquina sería recomendable crear una función por medio de la cual, si se pulsara la tecla BREAK, entonces aparecería un mensaje «L» y produciría un retorno al BASIC.

Cómo se provoca un mensaje de error

Consideremos ahora la posibilidad de provocar un mensaje de error. El Spectrum es capaz de entregar mensajes de error comprensibles para nosotros y, de hecho, no es demasiado difícil utilizarlos y generarlos a través de nuestro programa, aun cuando no son directamente aplicables al Código Máquina. Provocar un mensaje de error de esta forma no es demasiado difícil, especialmente cuando sabes cómo hacerlo pero, todavía a estas alturas, puede parecer un tanto confuso. En su lugar, utiliza un método que esté relacionado con el BASIC en vez de estarlo con el Código Máquina. Recuerda que si tratas de escribir el código de los 32 primeros caracteres se producirá un error. El motivo por el que sucede esto es porque estos caracteres son utilizados por el ordenador en instrucciones específicas para él mismo y que están relacionadas con el color, o con el final de una línea. Como ya sabes, se puede poner color en un programa simplemente pulsando CAPS SHIFT, SYMBOL SHIFT y luego

uno de los números. ¿Por qué no lo pruebas? Escribe PRINT CHR$ (13). Esta operación nos dará un mensaje de error. Resulta muy fácil utilizar este sistema en nuestro programa en Código Máquina cargando el registro C con un valor menor de 32 para luego volver al BASIC.

El breve programa en Código Máquina que se da a continuación utiliza esta propiedad en su funcionamiento y, para abreviar:

```
KEY-P EQUIVALE A 23557        (KEY-P = tecla pulsada)
LAST K EQUIVALE A 23560       (Recordar que LAST K = última
                               tecla)
INICIO    LD A, (KEY-P)
          SUB 5
          JR NZ, INICIO
¿BREAK?   LD A, (LAST K)
          SUB 127
          JR Z, SALIDA
          LD C, (LAST K)
          LD B, 0
          RET
SALIDA    LD C, 13
          LD B, 0
          RET
```

El programa comienza etiquetando las dos posiciones 23557 y 23560 con sus variables del sistema, KEY-P y LAST K. De esta forma, cuando nos remitamos a ellas en lo que queda de programa, podremos utilizar sus nombres en lugar de las direcciones respectivas. Esto nos facilita las cosas a la hora de teclear y a la hora de leerlas en un momento posterior. La rutina comienza cargando el registro A con el valor guardado en la variable del sistema KEY-P, luego le resta 5 (ya que si el resultado es 0 quiere decir que se ha pulsado una tecla). Esto es así porque, como recordarás, cuando se pulsa una tecla, se almacena el valor 5 en esta variable.

Si el resultado de restarle 5 al registro A es igual a 0 entonces es que se ha pulsado una tecla y si no es 0 la instrucción de salto relativo No-Cero (No-Zero) hará que se vuelva al INICIO y se lea de nuevo el valor de KEY-P y lo almacene en A.

De esta forma, permanecerá en un bucle sin fin hasta que sea pul-

sada una tecla y, como la variable del sistema KEY-P no nos dice qué tecla ha sido pulsada, ahora pasamos a comprobar la tecla BREAK. Esto se realiza casi de la misma forma que cuando comprobábamos si se había pulsado alguna tecla: cargando en el registro A el valor de la variable del sistema LAST K, y restándole el código de la tecla BREAK. Si se hubiera pulsado la tecla BREAK entonces el resultado sería 0, y el programa saltaría a la pequeña rutina de salida. Podemos observar que cuando se pulsa una tecla si no es la tecla BREAK, se cargará en el registro C el valor de la tecla pulsada, de forma que pueda ser escrita en la pantalla una vez que se vuelva al BASIC. Es importante que pongamos a 0 el contenido del registro B porque, cuando se vuelve al BASIC no se puede distinguir entre los dos registros B y C, y un número distinto del 0 en el registro B haría que el resultado fuera erróneo.

Una vez completada la operación, se vuelve al BASIC. La parte final del programa es una pequeña rutina llamada SALIDA, a la cual se accede sólo si se pulsa la tecla BREAK, en cuyo caso se carga el registro C con un valor no permitido y el registro B es puesto a 0. En ese caso, el ordenador volverá al BASIC. Como puede verse, la totalidad de la rutina puede colocarse en cualquier lugar y vale la pena grabarla en una cinta cassette para su posterior utilización bien en un programa BASIC bien en uno en Código Máquina.

Una vez analizado este proceso, ya podemos imaginarnos por qué no es posible «romper» o salirse de un programa en Código Máquina, especialmente en el caso de los programas comerciales, ya que a los autores de los mismos no les gusta que nadie vea lo que han escrito. Utilizando este método podrás resolver muchas de las dificultades que puedan presentarse.

La generación de caracteres y de gráficos de alta resolución

Ahora vamos a echar una ojeada al dispositivo principal de salida del ordenador —la pantalla—, conocido como el «dispositivo de salida» porque hace que la información «salga» para que la podamos observar. La propia pantalla está organizada de una forma tan maravillosa como misteriosa. Se utilizan 6K de memoria para almacenar la totalidad del patrón de puntos de la pantalla, es decir, para saber

si un punto determinado está (on) o no está (off) en la pantalla. Un K adicional de memoria sirve para almacenar los colores de cada carácter tanto si ese carácter está parpadeando (FLASH), si tiene más brillo (BRIGHT), etc. En este libro no se tratará de forma directa el tema del color de la pantalla sino que hablaremos sólo de la generación de caracteres y de gráficos de alta resolución.

Hay 256 puntos o «pixels» distribuidos horizontalmente a lo largo de la pantalla. Como el estado en que se encuentra cada «pixel» sólo puede ser «on» u «off», podemos registrar el estado de cada grupo de ocho de ellos dentro de un byte. Los ocho «pixels» son manipulados como si se tratara de un número binario por lo que si un «pixel» está en «on» cuenta como un 1 en notación binaria pero si está en «off» entonces cuenta como un 0 en notación binaria. El resultado es un número binario de ocho dígitos: 10101010.

Comprueba todo esto haciendo un POKE en el primer byte de la memoria de pantalla con el número binario citado anteriormente. Esto se realiza a través de la instrucción POKE 16384, BIN 10101010 (ENTER). Ahora trata de introducir combinaciones distintas de números binarios pero asegúrate siempre de no utilizar más de ocho cifras puesto que así se llegaría a un máximo de 255 en decimal (que es el máximo número que puede manejar el ordenador). Esta distribución rige en toda la pantalla 6144 veces. Esto se demuestra en el breve programa que se ofrece a continuación:

```
10 FOR A=16384 TO 16384+6143
20 POKE A,BIN 11111111
30 NEXT A
```

La división de la pantalla en tres secciones diferentes

Trata de cambiar el valor del número binario y se producirán algunos efectos muy interesantes. Te darás cuenta de que cuando llenamos la zona de la memoria de pantalla con un dígito concreto, ésta no se llena línea por línea de forma secuencial, desde la parte superior hasta la inferior de la pantalla. En su lugar, la pantalla se divide en tres secciones diferentes. Cada una de estas tres secciones está dividida en bloques de ocho caracteres, de arriba abajo. Primeramente

se llena la fila superior de cada bloque de caracteres, luego la segunda fila, y así continuamente hasta la fila octava inferior. A continuación este proceso vuelve a empezar en el bloque del centro y así hasta la fila inferior de los tres bloques. Esto puede parecer un tanto extraño y la verdad es que ¡lo es!

Otra forma de ilustrar esto es dibujando un diseño o una imagen en la pantalla y luego grabar la zona de la memoria de pantalla, la cual abarca desde la dirección 16384 hasta la dirección 22527, y, por último, borrar la pantalla y volver a cargarla. La zona de memoria de pantalla está situada en una parte concreta de la memoria, y como siempre se encuentra separada de la zona del BASIC o del programa en Código Máquina, la idea de cargar la pantalla con un dibujo al comienzo del programa a menudo es utilizada como una atractiva introducción ya que no utiliza ninguna parte de la memoria que, de otra forma, sería utilizada por el programa. Escribe el programa BASIC que aparece más abajo y luego grábalo a través del comando SAVE "" CODE 16384, 6144. Limpia la pantalla con el comando CLS y luego vuelve a cargar la pantalla con el comando LOAD "" CODE. El resultado será muy interesante.

```
10 LET N=631
20 LET A=120
25 PLOT 55,27: DRAW A,A,N*PI
```

La utilización de las rutinas en CM en un programa en BASIC

Veamos otro ejemplo. Carga en el ordenador las dos rutinas de Código Máquina para hacer el «scroll», «pixel» por «pixel», bien a la derecha bien a la izquierda. Escribe una breve rutina que llene la pantalla con el dibujo que acabas de grabar, y luego realiza repetidas veces el «scroll» a la izquierda. Después de una rotación completa de toda la pantalla, entonces vuelve a hacer que la pantalla haga el «scroll» a la derecha. No hay necesidad de escribir todo esto en Código Máquina. El resultado nos demostrará bastante claramente la utilización de las rutinas en Código Máquina dentro de un programa BASIC.

Como ya te habrás dado cuenta, debido a la extraña distribución de la pantalla, se necesita un programa muy enrevesado para tratar

de realizar un trabajo tal como el de escribir un carácter en la pantalla. De todas formas podemos concluir que, ya que el ordenador es capaz de hacerlo, entonces es necesario que exista una rutina dentro de la zona de ROM de la que podremos aprovecharnos. La importancia de esta rutina viene dada por el hecho de que esté situada muy cerca del principio de la memoria, empezando en la posición 16 (decimal) pero ¿cómo accedemos a esta rutina? Para ventaja nuestra ha sido colocada en una subrutina por lo que puede ser «llamada» (CALL).

Una característica del diseño del Z80 es la posibilidad que ofrece de poder acceder a algunas de las rutinas más importantes y las más utilizadas de una forma rápida y económica y por esto hay incorporado un juego de instrucciones que están destinadas a acceder a ocho de las primeras direcciones de la memoria. Quizás lo más importante de todo ello es que ¡sólo tienen una longitud de un byte! Una de estas rutinas, la utilizada para acceder a la rutina de escritura en la pantalla, es la «RST 16». Tiene el mismo efecto que «CALL 16» por lo que se necesita una instrucción RET al final de dicha rutina después de la cual el control volverá al comando inmediatamente posterior a RST 16.

Todavía hay que tener en cuenta un par de cosas antes de ejecutar este comando. Primero, la posición de escritura (PRINT) debe ser establecida por medio del BASIC y, aunque es posible hacerlo en Código Máquina, las cosas se complicarían innecesariamente. Segundo, el registro A debe ser cargado con el código del carácter que vamos a escribir. Esto es muy útil no sólo por la facilidad con que puede realizarse la conversión de un carácter y su código, utilizando la tabla ofrecida anteriormente sino también porque nos permite utilizar en el Código Máquina los Gráficos Definidos por el Usuario.

He aquí una sencilla rutina para escribir el carácter «A» en el extremo superior izquierdo de la pantalla. Acuérdate de acceder al Código Máquina en la forma ya conocida (en BASIC):

```
10 PRINT AT 0,0;
20 RANDOMIZE USR 30000


ORG 30000
```

```
30000 3E 61          LD A,97
30002 D7             RST 16
30003 C9             RET
```

Una rutina para crear los Gráficos Definidos por el Usuario

Ahora prueba a cambiar la posición de escritura y el contenido del registro A (el carácter que ha de escribirse). Evidentemente de aquí resulta fácil escribir un GDU (Gráfico Definido por el Usuario) cargando el registro A con el código apropiado. Ahora se necesita un método por el cual pueda representarse un carácter de GDU en Código Máquina. Desgraciadamente las rutinas utilizadas por el ordenador para hacer esto en el BASIC no son fácilmente accesibles desde el Código Máquina. Además, debido a que de cualquier forma se trata de un proceso complicado, te voy a ofrecer una breve rutina que te permitirá crear tus propios caracteres GDU. No es estrictamente necesario que a estas alturas comprendas su funcionamiento. Simplemente utilízala. Introdúcela y luego accede a ella a través de un comando CALL, seguida por nueve bytes de datos. El primero tendrá el número del carácter gráfico, es decir, tecla A o CHR$ 144 = 0, tecla B o CHR$ 145 = 1, y así sucesivamente. Esto permite especificar cuál es el carácter que va a ser definido. Los otros ocho bytes forman la «matriz de bits» y se utiliza del mismo modo que en la forma habitual y sirve para crear el carácter definido por el usuario.

La primera de las rutinas siguientes sirve para definir el carácter gráfico. La segunda es un ejemplo que sirve para definir los caracteres 144 y 145 (teclas A y B).

```
ORG 30000
EQU 65368 UDG
30000 0E 00          LD C,0
30002 E1             POP HL
30003 7E             LD A,(HL)
30004 A7             AND A
30005 87             ADD A,A
30006 87             ADD A,A
30007 87             ADD A,A
```

```
30008 11 58 FF      LD DE,UDG
30011 47            LD B,A
30012 7B            LD A,E
30013 80            ADD A,B
30014 5F            LD E,A
30015 7A            LD A,D
30016 06 00         LD B,0
30018 88            ADC A,B
30019 57            LD D,A
LOOP1
30020 79            LD A,C
30021 D6 08         SUB 0
30023 30 07         JR NC,EXIT
30025 0C            INC C
30026 23            INC HL
30027 7E            LD A,(HL)
30028 12            LD (DE),A
30030 18 F4         JR LOOP1
EXIT
30032 23            INC HL
30033 E5            PUSH HL
30034 C9            RET


ORG 28000
28000 CD 30 75      CALL C-GEN
DEFB 0
DEFB 1
DEFB 3
DEFB 7
DEFB 15
DEFB 31
DEFB 63
DEFB 127
DEFB 255
28012 CD 30 75      CALL C-GEN
DEFB 1
DEFB 255
DEFB 127
DEFB 63
```

```
DEFB 31
DEFB 15
DEFB 7
DEFB 3
DEFB 1
28024 C9                RET
```

No es necesario que entiendas todos los comandos que se necesitan para escribir un programa de Código Máquina que dibuje (PLOT), pero como resulta ser un programa un tanto difícil, a continuación se muestra la rutina que consigue dicho objetivo:

```
9C40  64        LD      H,H
9C41  64        LD      H,H
9C42  3A409C    LD      A,(9C40)
9C45  CB3F      SRL     A
9C47  CB3F      SRL     A
9C49  CB3F      SRL     A
9C4B  4F        LD      C,A
9C4C  3A419C    LD      A,(9C41)
9C4F  CB3F      SRL     A
9C51  CB3F      SRL     A
9C53  CB3F      SRL     A
9C55  47        LD      B,A
9C56  E600      AND     00
9C58  3E17      LD      A,17
9C5A  98        SBC     A,B
9C5B  D8        RET     C
9C5C  50        LD      D,B
9C5D  47        LD      B,A
9C5E  7A        LD      A,D
9C5F  CB27      SLA     A
9C61  CB27      SLA     A
9C63  CB27      SLA     A

9C65  57        LD      D,A
9C66  3A419C    LD      A,(9C41)
9C69  92        SUB     D
9C6A  57        LD      D,A
9C6B  F5        PUSH    AF
9C6C  78        LD      A,B
```

```
9C6D E618      AND    18
9C6F F640      OR     40
9C71 67        LD     H,A
9C72 F1        POP    AF
9C73 84        ADD    A,H
9C74 67        LD     H,A
9C75 78        LD     A,B
9C76 E607      AND    07
9C78 0F        RRCA
9C79 0F        RRCA
9C7A 0F        RRCA
9C7B 81        ADD    A,C
9C7C 6F        LD     L,A
9C7D 79        LD     A,C
9C7E CB27      SLA    A
9C80 CB27      SLA    A
9C82 CB27      SLA    A
9C84 47        LD     B,A
9C85 3A409C    LD     A,(9C40)
9C88 90        SUB    B
9C89 FE00      CP     00
9C8B 281D      JR     Z,9CAA
9C8D FE01      CP     01
9C8F 281C      JR     Z,9CAD
9C91 FE02      CP     02
9C93 281B      JR     Z,9CB0
9C95 FE03      CP     03
9C97 281A      JR     Z,9CB3
9C99 FE04      CP     04
9C9B 2819      JR     Z,9CB6
9C9D FE05      CP     05
9C9F 2818      JR     Z,9CB9
9CA1 FE06      CP     06
9CA3 2817      JR     Z,9CBC
9CA5 FE07      CP     07
9CA7 2816      JR     Z,9CBF
9CA9 C9        RET
9CAA CBFE      BIT    7,(HL)
9CAC C9        RET
9CAD CBF6      BIT    6,(HL)
9CAF C9        RET
```

```
9CB0 CBEE      BIT     5,(HL)
9CB2 C9        RET
9CB3 CBE6      BIT     4,(HL)
9CB5 C9        RET
9CB6 CBDE      BIT     3,(HL)
9CB8 C9        RET
9CB9 CBD6      BIT     2,(HL)
9CBB C9        RET
9CBC CBCE      BIT     1,(HL)
9CBE C9        RET
9CBF CBC6      BIT     0,(HL)
9CC1 C9        RET
9CC2 00        NOP
9CC3 00        NOP
9CC4 00        NOP
9CC5 00        NOP
```

Un objetivo primordial: profundizar en el CM

Capítulo 10

LA PILA DE MEMORIA (STACK)

Este capítulo estará dedicado a analizar el funcionamiento de otros comandos de Código Máquina, a ver algunos nuevos conceptos y a estudiar una zona muy útil empleada por la propia CPU. Será necesario repasar las instrucciones INC y DEC lo que nos llevará a las instrucciones LDIR y LDDR. Conocer totalmente estas últimas instrucciones exige principalmente una familiarización con las instrucciones INC y DEC.

Quizá sea ésta la primera vez que vayamos a manejar un término que significa exactamente lo que expresa. La función de la pila es la de comportarse como una pila y nada más que como una pila. La pila (stack) sirve para apilar números. Se puede poner un número en lo alto (o parte superior) y luego quitarlo de allí. Sólo se puede quitar de la parte superior, nunca de ningún otro sitio.

Imagínatela como si fuera una torre de cajas. Es posible retirar la caja que se encuentra en la parte más alta o, en su lugar, poner otra en la parte superior de dicha pila pero ¡no intentes hacer otra cosa por tu cuenta! Esto es exactamente lo que pasa en Código Máquina. La única diferencia es que cada caja está numerada con una dirección. Aunque una de las ventajas de la pila es que no es necesario que nos preocupemos de las direcciones.

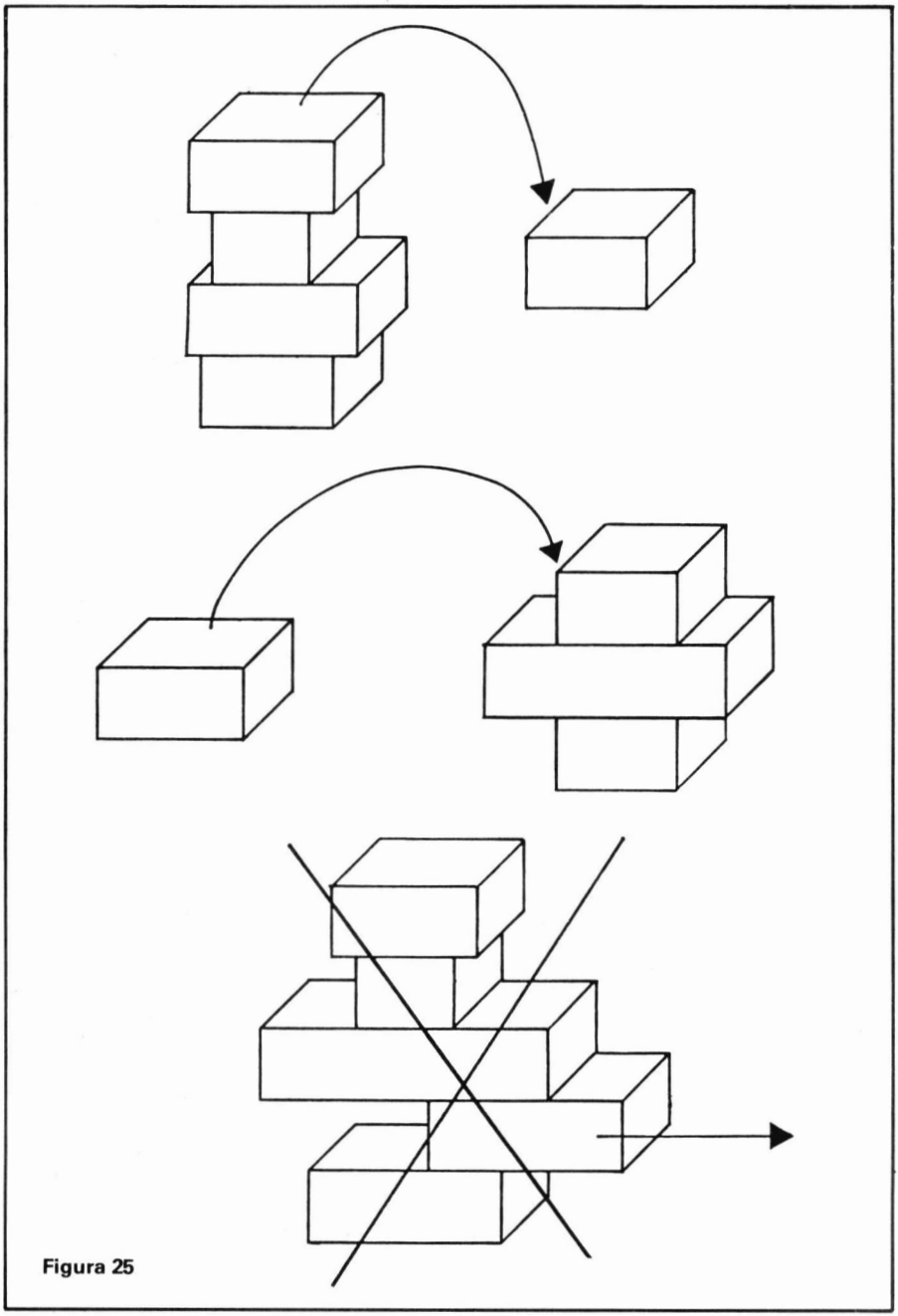


Figura 25

El «puntero de pila»

En principio, no hay ninguna restricción en el tamaño de una pila aunque, a medida que se hace mayor, nos quedará menos sitio para otras cosas, además de la posibilidad de escribir por encima de otros códigos. El establecimiento de una pila es una tarea muy sencilla: consiste en decidir en qué posición de la memoria debe comenzar y luego asignar ese valor a la variable del sistema STK-P. En efecto, esto crea lo que se llama un **puntero de la pila** (Fig. 25).

El puntero de la pila «apunta» a la posición de la memoria en la que se va a colocar el número siguiente.

```
ORG 30000              (EQU=EQUIVALENCIA)
EQU 28000 STK-P
30000 3E 10            LD A,16
30002 FD 22 60 6D      LD (STK-P),16
30006 21 60 6D         LD HL,STK-P
30009 35               DEC (HL)
```

Sin un ensamblador, este proceso resulta un poco más difícil. Primero elige una posición que pueda ser utilizada para almacenar el puntero de la pila, luego asegúrate de que el número en su interior es el 0, carga el registro doble HL con esta posición y luego carga la posición con el valor del puntero de la pila.

```
30010 3E 10            LD A,16
30012 21 60 6D         LD HL,28000
30015 77               LD (HL),A
30016 2B               DEC HL
```

Cuando se coloca un número en la cima de la pila tenemos que disminuir en 1 el valor del puntero, de forma que la próxima vez que se añada otro número a la pila no sólo no se superpondrá al primero sino que será colocado en una posición «menos 1». Antes de poner un número en la pila, hay que introducirlo en el registro A. Luego es cargado en la posición señalada por el puntero de la pila y el valor de dicho puntero es disminuido en 1 (Fig. 26).

Lo único que falta es crear una rutina que devuelva ese número

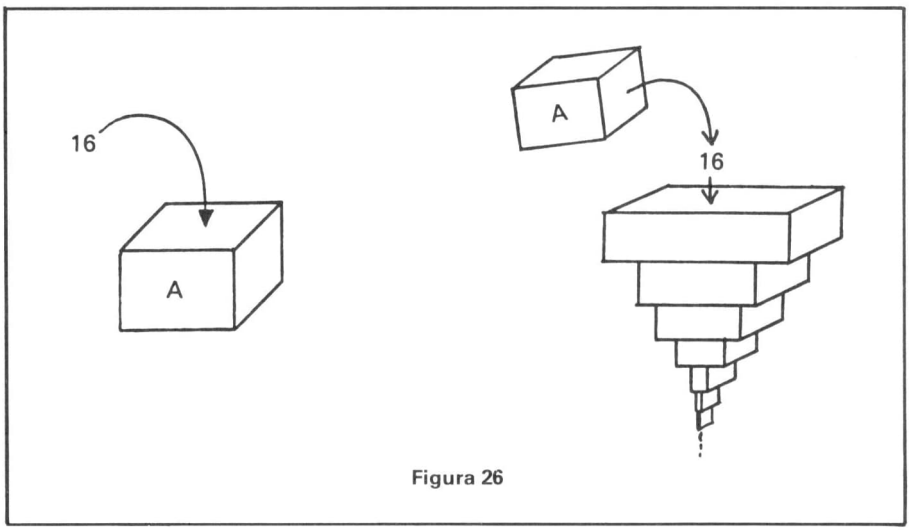


Figura 26

que está en la pila. En este caso, carga el registro A con el número almacenado de la cima de la pila y luego incrementa en 1 el valor guardado por el puntero de la pila (Fig. 27).

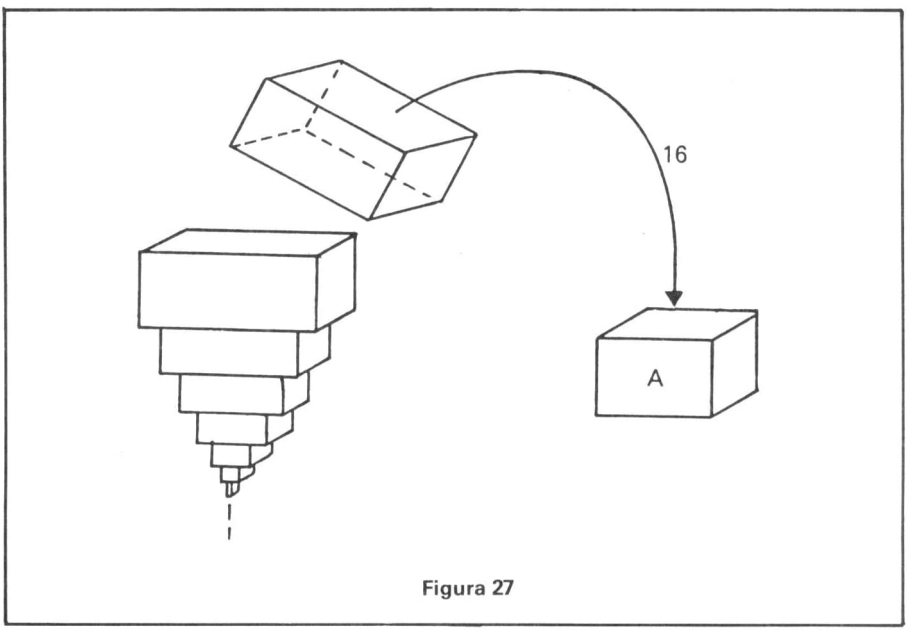


Figura 27

La utilización de una pila dentro de un programa

Este programa nos permite crear y utilizar una pila bastante sencilla. La idea de tener una pila de números —de la cual podemos tomar números o a la cual podemos añadírselos, puede parecer un tanto consumidora de espacio pero la utilización de una pila dentro de un programa, empleando las rutinas anteriores, en la práctica, puede resultar sumamente útil y puede ser una excelente forma de simplificar un programa. Probablemente el método más sencillo de utilizar una pila, y que resulta más fácil de volver a leer, es el de etiquetar las dos subrutinas: la que sirve para colocar un número en la pila y la que lo devuelve para ser utilizado de nuevo —con PUSH (empujar) y POP (meter)— y luego acceder a ellas utilizando las instrucciones CALL. En este programa se simulan estas instrucciones de Código Máquina, las cuales harían el trabajo de otra forma pero su utilización es mucho más complicada.

```
ORG 30000
EQU 28000 STK-P
30000 3A 30 75        LD A,(STK-P)
30003 21 30 75        LD HL,STK-P
30006 34              INC (HL)
30007 21 60 6D        LD HL,28000
30010 7E              LD A,(HL)
30011 23              INC HL
```

El juego de «Las Torres de Hanoi»

Hagamos un pequeño intermedio. El programa BASIC, que se ofrece a continuación, trata de un juego que funciona utilizando el mismo principio que la propia pila. Aunque no está escrito en Código Máquina, la idea del juego está directamente relacionada con este capítulo, y más directamente con el funcionamiento de la pila. El fundamento del juego es muy antiguo y puede que ya lo hayas visto con anterioridad en cualquier otro lado. Se llama «Las Torres de Hanoi» y el objetivo del juego consiste en trasladar una pila de cinco anillos de tamaño creciente desde una clavija o torre hasta la clavija o torre final. Pero hay un problema que consiste en que

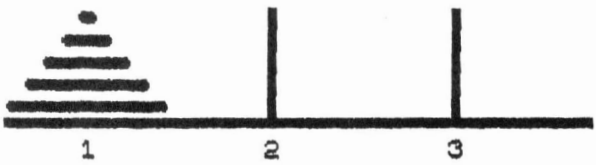
los anillos deben seguir en el mismo orden ascendente con el que se empezó. Además sólo se puede trasladar un anillo cada vez y deberá ser colocado en lo alto de cada pila. No se puede colocar un anillo encima de otro si es más pequeño que él. Esto te dará una pequeña idea de los problemas que hay que tener en cuenta cuando se trabaja con las pilas, ¡especialmente si pierdes el rastro de lo que se encontraba en su cima!

```
   2 GO SUB 2000
   5 BORDER 6: PAPER 6: INK 0: C
LS
   6 REM GRAFICOS LINEA SIGUIEN-
TE; 1- E , 2- ABC , 3- ABBBC , 4
- ABBBBBC , 5- ABBBBBBBC , 0- D
   7 DATA "1     ▬    ","2    ▬▬
 ","3  ▬▬▬▬  ","4 ▬▬▬▬▬▬  ","5
▬▬▬▬▬▬▬▬▬▬","0    D     "
  10 RESTORE 7: LET C=10: LET M=
0: DIM A$(16,11): DIM C$(5,11):
LET W=0: DIM B$(2,1): FOR D=1 TO
 16
  20 IF D<7 THEN  READ A$(D): GO
 TO 40
  30 LET A$(D)=A$(6)
  40 NEXT D
 400 FOR X=1 TO 5: LET C$(X)=A$(
X): NEXT X
 470 PRINT AT 0,7;"TORRES DE HAN
OI"
 475 PRINT AT 20,22;M;" MOVS."
 480 FOR D=0 TO 31: PRINT AT 16,
D;"▬": NEXT D: PRINT AT 17,4; IN
K 1;"1";TAB 14;"2";TAB 24;"3"
 500 FOR B=1 TO 15: LET C=C+1: P
RINT AT C,INT (B/5-.1)*10;A$(B,2
 TO )
 510 IF C=15 THEN  LET C=10
 520 NEXT B
 530 IF W=1 THEN  GO TO 1600
```

```
 580 INPUT "DEL POSTE? "; LINE B
$(1),"AL POSTE? "; LINE B$(2): P
RINT AT 21,10;"
": IF B$(1)<STR$ 1 OR B$(1)>STR$
 3 OR B$(2)<STR$ 1 OR B$(2)>STR$
 3 THEN  GO TO 1500
 600 FOR Z=1 TO 5
 610 IF A$((VAL B$(1)-1)*5+Z)<>A
$(16) THEN  GO TO 640
 615 IF Z=5 AND A$((VAL B$(1)-1)
*5+Z)=A$(16) THEN  GO TO 1500
 620 NEXT Z
 640 FOR Y=5 TO 1 STEP -1
 660 IF A$((VAL B$(2)-1)*5+Y)=A$
(16) THEN  GO TO 1000
 680 NEXT Y
1000 IF Y=5 THEN  GO TO 1010
1003 IF A$((VAL B$(2)-1)*5+Y+1)<
A$((VAL B$(1)-1)*5+Z) THEN  GO T
O 1500
1010 LET A$((VAL B$(2)-1)*5+Y)=A
$((VAL B$(1)-1)*5+Z)
1020 LET A$((VAL B$(1)-1)*5+Z)=A
$(16)
1030 LET M=M+1
1040 FOR D=1 TO 5: IF A$(D+10)<>
C$(D) THEN  GO TO 470
1050 NEXT D: LET W=1: GO TO 470
1500 BEEP 1,-20: PRINT AT 21,10;
 FLASH 1;"MOVIMIENTO ILEGAL"
1520 GO TO 580
1600 PRINT AT 5,11; FLASH 1;"BIE
N HECHO"; FLASH 0;''TAB 2;"LO HI
CISTE EN ";M;" MOVIMIENTOS,": IF
 M=31 THEN  PRINT TAB 2; FLASH 1
;"!!!LO CUAL ES INMEJORABLE!!!":
 FLASH 0: FOR X=0 TO 255: OUT 25
4,X: NEXT X: GO TO 1605
1601 PRINT TAB 6;"PERO RUEDE MEJ
ORARSE."
```

```
1610 INPUT "OTRO JUEGO?-S/N";Q$:
 IF Q$<>"S" AND Q$<>"S" THEN  PR
INT AT 21,0;"ADIOS,,,": STOP
1620 RUN
2000 RESTORE 3000
2005 FOR Y=0 TO 5
2010 FOR X=0 TO 7
2020 READ CHR: POKE 65368+Y*8+X,
CHR
2030 NEXT X
2040 NEXT Y
2050 RETURN
3000 DATA 0,0,BIN 00111111,BIN 0
1111111,BIN 01111111,BIN 0011111
1,0,0
3010 DATA 0,0,255,255,255,255,0,
0
3020 DATA 0,0,BIN 11111100,BIN 1
1111110,BIN 11111110,BIN 1111110
0,0,0
3030 DATA 60,60,60,60,60,60,60,6
0
3040 DATA 0,24,60,60,60,60,60,60
3050 DATA 0,0,BIN 1111110,255,25
5,BIN 1111110,0,0
```

TORRES DE HANOI



0 MOVS.

El acceso a una pila del BASIC a través del Código Máquina

El BASIC, directa e indirectamente, utiliza tres tipos distintos de pilas para sus operaciones: la pila de GOSUB, que almacena las líneas de retorno de todas las subrutinas del BASIC; la pila del calculador, que se utiliza para todas las operaciones numéricas del BASIC; finalmente la pila del ordenador, la cual es utilizada por la propia CPU y que es accesible a través del Código Máquina. Pero colocar algo o utilizar la pila del ordenador desde el BASIC no resulta nada fácil, y siempre es conveniente ir al Código Máquina antes de utilizarla.

El traslado del contenido de un bloque de memoria a otro bloque

Anteriormente ya hemos visto las instrucciones INC y DEC y cómo hacen que el contenido de un registro o de una posición sea INCrementado o DECrementado (disminuido) en uno. Hay algunas utilizaciones claras en las que encajan perfectamente. Un ejemplo de ello consiste en trasladar el contenido de un bloque de memoria a otro bloque. En este caso utilizaremos dos punteros para señalar esas dos zonas de la memoria las cuales, por comodidad, podemos considerar como el Bloque Fuente y el Bloque de Destino. Los comandos INC y DEC son utilizados para incrementar o disminuir los valores almacenados en los punteros y evitar de esta forma que sean recargados antes de realizar cada traslado. Por ejemplo:

```
LD HL, FUENTE
LD DE, DESTINO
LD A, (HL)
LD (DE), A
INC HL
INC DE
LD A, (HL)
LD (DE), A
INC HL
INC DE
INC DE
INC DE
```

La utilización del bucle en los traslados

Esto podría servir para trasladar uno o dos bytes pero no para cualquier otra cosa más ambiciosa pues sería demasiado laborioso. Afortunadamente el BUCLE viene a salvarnos, pero antes de utilizar este comando debemos especificar, dentro de un registro, el número de «veces» que se necesita ejecutar dicho bucle. Esto se hace de la forma siguiente:

```
LD HL, FUENTE
LD DE, DESTINO
LD BC, NUMERO DE VECES
BUCLE
LD A, (HL)
LD (DE), A
INC DE
INC HL
DEC BC
LD A, B
ADD A, C
JR NZ, BUCLE
RET
```

Simplificación de rutinas utilizando los comandos «INC», «DEC», «LDIR» y «LDDR»

Sumando el contenido del byte más significativo (B) al del byte menos significativo (C) nos es posible comprobar si el número almacenado en el registro es un cero o no. Todo esto nos proporciona, además, un medio relativamente sencillo con el que podemos desplazar un bloque de memoria.

La rutina anterior puede simplificarse más utilizando dos comandos que sólo requieren dos bytes cada uno (INC y DEC sólo necesitan un byte cada uno), hasta tal punto que podemos eliminar todos los comandos de la rutina menos cuatro. Estos dos nuevos comandos son «LDIR» y «LDDR», y sus definiciones son:

LDIR = «Transferencia condicional de memoria-a-memoria con auto-incremento de los registros punteros de la memoria, y auto-disminución del registro contador de bytes».

LDDR = «Transferencia condicional de memoria-a-memoria con auto-disminución de los registros punteros de la memoria, y auto-disminución del registro contador de bytes».

Todo esto quiere decir que, con el comando LDIR podemos cargar el registro doble HL con el comienzo del bloque fuente; el registro doble DE con el comienzo de la zona de destino de la memoria, y el registro doble BC con el número de bytes que van a ser trasladados. Por lo tanto, con la utilización del comando LDIR, el número de bytes especificados (en BC) son desplazados desde la posición HL a la posición DE, incrementando cada vez los valores de HL y DE. El valor de BC será disminuido en uno cada vez hasta que alcance el cero, lo cual pondrá fin al comando y el ordenador pasará al comando siguiente. El comando LDDR precisamente funciona de la misma forma, excepto, por supuesto, que esta vez se disminuye el contenido de los dos punteros en vez de incrementarse. Una rutina que utilizara estos dos comandos tendría el siguiente aspecto:

```
LD HL, FUENTE
LD DE, DESTINO
LD BC, BYTES
LDIR
RET
```

Una rutina para desplazar todo el contenido de la pantalla

Por ejemplo, una rutina que desplazara todo el contenido de la pantalla tendría este aspecto:

```
ORG 25501
25501 21 00 40      LD HL,16384
25504 11 00 64      LD DE,25600
25507 01 00 1C      LD BC,7168
25510 ED B0         LDIR
```

```
25512 C9          RET
25513 21 00 64    LD HL,25600
25516 11 00 40    LD DE,16384
25519 01 00 1C    LD BC,7168
25522 ED B0       LDIR
25524 C9          RET
```

Un comando de velocidad increíble

Ahora escribe LOAD en el programa BASIC de «dibujo» dado anteriormente, almacena la pantalla utilizando la primera de las dos rutinas anteriores, borra la pantalla (CLS) y traslada de nuevo el dibujo por medio de la segunda rutina. Fíjate en la velocidad increíble de este comando.

Esto nos lleva al final de esta etapa a lo largo de los fundamentos de la programación en Código Máquina, una etapa que confío que haya sido tan agradable como informativa. Si de verdad te ha dejado con el deseo de analizar con más profundidad en este tema, entonces habrá valido la pena. Si te ha dejado en una situación lo bastante buena como para utilizar la información de otros libros más avanzados, entonces sí que habré triunfado.

BUENA PROGRAMACION

J. H. C. W.

APENDICE A

00	0	00000000
01	1	00000001
02	2	00000010
03	3	00000011
04	4	00000100
05	5	00000101
06	6	00000110
07	7	00000111
08	8	00001000
09	9	00001001
0A	10	00001010
0B	11	00001011
0C	12	00001100
0D	13	00001101
0E	14	00001110
0F	15	00001111
10	16	00010000
11	17	00010001
12	18	00010010
13	19	00010011
14	20	00010100
15	21	00010101
16	22	00010110
17	23	00010111
18	24	00011000
19	25	00011001
1A	26	00011010
1B	27	00011011
1C	28	00011100
1D	29	00011101

1E	30	00011110
1F	31	00011111
20	32	00100000
21	33	00100001
22	34	00100010
23	35	00100011
24	36	00100100
25	37	00100101
26	38	00100110
27	39	00100111
28	40	00101000
29	41	00101001
2A	42	00101010
2B	43	00101011
2C	44	00101100
2D	45	00101101
2E	46	00101110
2F	47	00101111
30	48	00110000
31	49	00110001
32	50	00110010
33	51	00110011
34	52	00110100
35	53	00110101
36	54	00110110
37	55	00110111
38	56	00111000
39	57	00111001
3A	58	00111010
3B	59	00111011
3C	60	00111100
3D	61	00111101
3E	62	00111110
3F	63	00111111
40	64	01000000
41	65	01000001
42	66	01000010
43	67	01000011
44	68	01000100
45	69	01000101
46	70	01000110

47	71	01000111
48	72	01001000
49	73	01001001
4A	74	01001010
4B	75	01001011
4C	76	01001100
4D	77	01001101
4E	78	01001110
4F	79	01001111
50	80	01010000
51	81	01010001
52	82	01010010
53	83	01010011
54	84	01010100
55	85	01010101
56	86	01010110
57	87	01010111
58	88	01011000
59	89	01011001
5A	90	01011010
5B	91	01011011
5C	92	01011100
5D	93	01011101
5E	94	01011110
5F	95	01011111
60	96	01100000
61	97	01100001
62	98	01100010
63	99	01100011
64	100	01100100
65	101	01100101
66	102	01100110
67	103	01100111
68	104	01101000
69	105	01101001
6A	106	01101010
6B	107	01101011
6C	108	01101100
6D	109	01101101
6E	110	01101110
6F	111	01101111

70	112	01110000
71	113	01110001
72	114	01110010
73	115	01110011
74	116	01110100
75	117	01110101
76	118	01110110
77	119	01110111
78	120	01111000
79	121	01111001
7A	122	01111010
7B	123	01111011
7C	124	01111100
7D	125	01111101
7E	126	01111110
7F	127	01111111
80	128	10000000
81	129	10000001
82	130	10000010
83	131	10000011
84	132	10000100
85	133	10000101
86	134	10000110
87	135	10000111
88	136	10001000
89	137	10001001
8A	138	10001010
8B	139	10001011
8C	140	10001100
8D	141	10001101
8E	142	10001110
8F	143	10001111
90	144	10010000
91	145	10010001
92	146	10010010
93	147	10010011
94	148	10010100
95	149	10010101
96	150	10010110
97	151	10010111
98	152	10011000

99	153	10011001
9A	154	10011010
9B	155	10011011
9C	156	10011100
9D	157	10011101
9E	158	10011110
9F	159	10011111
A0	160	10100000
A1	161	10100001
A2	162	10100010
A3	163	10100011
A4	164	10100100
A5	165	10100101
A6	166	10100110
A7	167	10100111
A8	168	10101000
A9	169	10101001
AA	170	10101010
AB	171	10101011
AC	172	10101100
AD	173	10101101
AE	174	10101110
AF	175	10101111
B0	176	10110000
B1	177	10110001
B2	178	10110010
B3	179	10110011
B4	180	10110100
B5	181	10110101
B6	182	10110110
B7	183	10110111
B8	184	10111000
B9	185	10111001
BA	186	10111010
BB	187	10111011
BC	188	10111100
BD	189	10111101
BE	190	10111110
BF	191	10111111
C0	192	11000000
C1	193	11000001

C2	194	11000010
C3	195	11000011
C4	196	11000100
C5	197	11000101
C6	198	11000110
C7	199	11000111
C8	200	11001000
C9	201	11001001
CA	202	11001010
CB	203	11001011
CC	204	11001100
CD	205	11001101
CE	206	11001110
CF	207	11001111
D0	208	11010000
D1	209	11010001
D2	210	11010010
D3	211	11010011
D4	212	11010100
D5	213	11010101
D6	214	11010110
D7	215	11010111
D8	216	11011000
D9	217	11011001
DA	218	11011010
DB	219	11011011
DC	220	11011100
DD	221	11011101
DE	222	11011110
DF	223	11011111
E0	224	11100000
E1	225	11100001
E2	226	11100010
E3	227	11100011
E4	228	11100100
E5	229	11100101
E6	230	11100110
E7	231	11100111
E8	232	11101000
E9	233	11101001
EA	234	11101010

EB	235	11101011
EC	236	11101100
ED	237	11101101
EE	238	11101110
EF	239	11101111
F0	240	11110000
F1	241	11110001
F2	242	11110010
F3	243	11110011
F4	244	11110100
F5	245	11110101
F6	246	11110110
F7	247	11110111
F8	248	11111000
F9	249	11111001
FA	250	11111010
FB	251	11111011
FC	252	11111100
FD	253	11111101
FE	254	11111110
FF	255	11111111

Programa para realizar el listado de las páginas precedentes:

```
  10 FOR A=0 TO 255
  20 GO SUB 400: LPRINT "        "
;A;"        ";: GO SUB 200: LPRINT

  30 NEXT A
 200 LET N=A
 210 FOR X=7 TO 0 STEP -1
 220 IF INT (N/(2↑X))=1 THEN  GO
TO 300
 230 LPRINT 0;
 240 NEXT X
 250 RETURN
 300 LPRINT 1;
 310 LET N=N-2↑X
 320 GO TO 240
```

```
400 LET N=A
410 LET X=INT (N/16)
420 LET Y=((N/16)-INT (N/16))*1
6
430 IF X>9 THEN  LET X=X+7
440 IF Y>9 THEN  LET Y=Y+7
450 LET X=X+48: LET Y=Y+48
460 LPRINT CHR$ X;CHR$ Y;
470 RETURN
```

APENDICE B

CODIGOS DE OPERACION DEL Z80 CLASIFICADOS POR NEMÓNICOS

Código de operación		Hex	Decimal
ADC	A,(HL)	8E	142
ADC	A,(IX+d)	DD8Edd	221,142,XX
ADC	A,(IY+d)	FD8Edd	253,142,XX
ADC	A,A	8F	143
ADC	A,B	88	136
ADC	A,C	89	137
ADC	A,D	8A	138
ADC	A,E	8B	139
ADC	A,H	8C	140
ADC	A,L	8D	141
ADC	A,xx	CExx	206,XX
ADC	HL,BC	ED4A	237,74
ADC	HL,DE	ED5A	237,90
ADC	HL,HL	ED6A	237,106
ADC	HL,SP	ED7A	237,122
ADD	A,(HL)	86	134
ADD	A,(IX+d)	DD86dd	221,134,XX
ADD	A,(IY+d)	FD86dd	253,134,XX
ADD	A,A	87	135
ADD	A,B	80	128
ADD	A,C	81	129
ADD	A,D	82	130
ADD	A,E	83	131
ADD	A,H	84	132
ADD	A,L	85	133
ADD	A,xx	C6xx	198,XX
ADD	HL,BC	09	9
ADD	HL,DE	19	25

Codigo de operación		Hex	Decimal
ADD	HL,HL	29	41
ADD	HL,SP	39	57
ADD	IX,BC	DD09	221,9
ADD	IX,DE	DD19	221,25
ADD	IX,IX	DD29	221,41
ADD	IX,SP	DD39	221,57
ADD	IY,BC	FD09	253,9
ADD	IY,DE	FD19	253,25
ADD	IY,IY	FD29	253,41
ADD	IY,SP	FD39	253,57
AND	(HL)	A6	166
AND	(IX+d)	DDA6dd	221,166,XX
AND	(IY+d)	FDA6dd	253,166,XX
AND	A	A7	167
AND	B	A0	160
AND	C	A1	161
AND	D	A2	162
AND	E	A3	163
AND	H	A4	164
AND	L	A5	165
AND	xx	E6xx	230,XX
BIT	0,(HL)	CB46	203,70
BIT	0,(IX+d)	DDCBdd46	221,203,XX,70
BIT	0,(IY+d)	FDCBdd46	253,203,XX,70
BIT	0,A	CB47	203,71
BIT	0,B	CB40	203,64
BIT	0,C	CB41	203,65
BIT	0,D	CB42	203,66
BIT	0,E	CB43	203,67
BIT	0,H	CB44	203,68
BIT	0,L	CB45	203,69
BIT	1,(HL)	CB4E	203,78
BIT	1,(IX+d)	DDCBdd4E	221,203,XX,78
BIT	1,(IY+d)	FDCBdd4E	253,203,XX,78
BIT	1,A	CB4F	203,79
BIT	1,B	CB48	203,72
BIT	1,C	CB49	203,73
BIT	1,D	CB4A	203,74
BIT	1,E	CB4B	203,75
BIT	1,H	CB4C	203,76
BIT	1,L	CB4D	203,77
BIT	2,(HL)	CB56	203,86
BIT	2,(IX+d)	DDCBdd56	221,203,XX,86
BIT	2,(IY+d)	FDCBdd56	253,203,XX,86
BIT	2,A	CB57	203,87

Código de operación	Hex	Decimal
BIT 2,B	CB50	203,80
BIT 2,C	CB51	203,81
BIT 2,D	CB52	203,82
BIT 2,E	CB53	203,83
BIT 2,H	CB54	203,84
BIT 2,L	CB55	203,85
BIT 3,(HL)	CB5E	203,94
BIT 3,(IX+d)	DDCBdd5E	221,203,XX,94
BIT 3,(IY+d)	FDCBdd5E	253,203,XX,94
BIT 3,A	CB5F	203,95
BIT 3,B	CB58	203,88
BIT 3,C	CB59	203,89
BIT 3,D	CB5A	203,90
BIT 3,E	CB5B	203,91
BIT 3,H	CB5C	203,92
BIT 3,L	CB5D	203,93
BIT 4,(HL)	CB66	203,102
BIT 4,(IX+d)	DDCBdd66	221,203,XX,102
BIT 4,(IY+d)	FDCBdd66	253,203,XX,102
BIT 4,A	CB67	203,103
BIT 4,B	CB60	203,96
BIT 4,C	CB61	203,97
BIT 4,D	CB62	203,98
BIT 4,E	CB63	203,99
BIT 4,H	CB64	203,100
BIT 4,L	CB65	203,101
BIT 5,(HL)	CB6E	203,110
BIT 5,(IX+d)	DDCBdd6E	221,203,XX,110
BIT 5,(IY+d)	FDCBdd6E	253,203,XX,110
BIT 5,A	CB6F	203,111
BIT 5,B	CB68	203,104
BIT 5,C	CB69	203,105
BIT 5,D	CB6A	203,106
BIT 5,E	CB6B	203,107
BIT 5,H	CB6C	203,108
BIT 5,L	CB6D	203,109
BIT 6,(HL)	CB76	203,118
BIT 6,(IX+d)	DDCBdd76	221,203,XX,118
BIT 6,(IY+d)	FDCBdd76	253,203,XX,118
BIT 6,A	CB77	203,119
BIT 6,B	CB70	203,112
BIT 6,C	CB71	203,113
BIT 6,D	CB72	203,114
BIT 6,E	CB73	203,115
BIT 6,H	CB74	203,116
BIT 6,L	CB75	203,117

Código de operación	Hex	Decimal
BIT 7,(HL)	CB7E	203,126
BIT 7,(IX+d)	DDCBdd7E	221,203,XX,126
BIT 7,(IY+d)	FDCBdd7E	253,203,XX,126
BIT 7,A	CB7F	203,127
BIT 7,B	CB78	203,120
BIT 7,C	CB79	203,121
BIT 7,D	CB7A	203,122
BIT 7,E	CB7B	203,123
BIT 7,H	CB7C	203,124
BIT 7,L	CB7D	203,125
CALL C,xxxx	DCxxxx	220,XX,XX
CALL M,xxxx	FCxxxx	252,XX,XX
CALL NC,xxxx	D4xxxx	212,XX,XX
CALL NZ,xxxx	C4xxxx	196,XX,XX
CALL P,xxxx	F4xxxx	244,XX,XX
CALL PE,xxxx	ECxxxx	236,XX,XX
CALL PO,xxxx	E4xxxx	228,XX,XX
CALL xxxx	CDxxxx	205,XX,XX
CALL Z,xxxx	CCxxxx	204,XX,XX
CCF	3F	63
CP (HL)	BE	190
CP (IX+d)	DDBEdd	221,190,XX
CP (IY+d)	FDBEdd	253,190,XX
CP A	BF	191
CP B	B8	184
CP C	B9	185
CP D	BA	186
CP E	BB	187
CP H	BC	188
CP L	BD	189
CP xx	FExx	254,XX
CPD	EDA9	237,169
CPDR	EDB9	237,185
CPI	EDA1	237,161
CPIR	EDB1	237,177
CPL	2F	47
DAA	27	39
DEC (HL)	35	53
DEC (IX+d)	DD35dd	221,53,XX
DEC (IY+d)	FD35dd	253,53,XX
DEC A	3D	61
DEC B	05	5
DEC BC	0B	11
DEC C	0D	13
DEC D	15	21
DEC DE	1B	27

DEC	E	1D	29
DEC	H	25	37
DEC	HL	2B	43
DEC	IX	DD2B	221,43
DEC	IY	FD2B	253,43
DEC	L	2D	45
DEC	SP	3B	59
DI		F3	243
DJNZ	xx	10xx	16,XX
EI		FB	251
EX	(SP),HL	E3	227
EX	(SP),IX	DDE3	221,227
EX	(SP),IY	FDE3	253,227
EX	AF,AF	08	8
EX	DE,HL	EB	235
Exx		D9	217
HALT		76	118
IM	0	ED46	237,70
IM	1	ED56	237,86
IM	2	ED5E	237,94
IN	A,(C)	ED78	237,120
IN	A,(xx)	DBxx	219,XX
IN	B,(C)	ED40	237,64
IN	C,(C)	ED48	237,72
IN	D,(C)	ED50	237,80
IN	E,(C)	ED58	237,88
IN	H,(C)	ED60	237,96
IN	L,(C)	ED68	237,104
INC	(HL)	34	52
INC	(IX+d)	DD34dd	221,52,XX
INC	(IY+d)	FD34dd	253,52,XX
INC	A	3C	60
INC	B	04	4
INC	BC	03	3
INC	C	0C	12
INC	D	14	20
INC	DE	13	19
INC	E	1C	28
INC	H	24	36
INC	HL	23	35
INC	IX	DD23	221,35
INC	IY	FD23	253,35
INC	L	2C	44
INC	SP	33	51
IND		EDAA	237,170

INDR		EDBA	237,186
INI		EDA2	237,162
INIR		EDB2	237,178
JP	(HL)	E9	233
JP	(IX)	DDE9	221,233
JP	(IY)	FDE9	253,233
JP	C,xxxx	DAxxxx	218,XX,XX
JP	M,xxxx	FAxxxx	250,XX,XX
JP	NC,xxxx	D2xxxx	210,XX,XX
JP	NZ,xxxx	C2xxxx	194,XX,XX
JP	P,xxxx	F2xxxx	242,XX,XX
JP	PE,xxxx	EAxxxx	234,XX,XX
JP	PO,xxxx	E2xxxx	226,XX,XX
JP	xxxx	C3xxxx	195,XX,XX
JP	Z,xxxx	CAxxxx	202,XX,XX
JR	C,xx	38xx	56,XX
JR	NC,xx	30xx	48,XX
JR	NZ,xx	20xx	32,XX
JR	xx	18xx	24,XX
JR	Z,xx	28xx	40,XX
LD	(BC),A	02	2
LD	(DE),A	12	18
LD	HL,(xxxx)	2Axxxx	42,XX,XX
LD	(HL),A	77	119
LD	(HL),B	70	112
LD	(HL),C	71	113
LD	(HL),D	72	114
LD	(HL),E	73	115
LD	(HL),H	74	116
LD	(HL),L	75	117
LD	(HL),xx	36xx	54,XX
LD	(IX+d),A	DD77dd	221,119,XX
LD	(IX+d),B	DD70dd	221,112,XX
LD	(IX+d),C	DD71dd	221,113,XX
LD	(IX+d),D	DD72dd	221,114,XX
LD	(IX+d),E	DD73dd	221,115,XX
LD	(IX+d),H	DD74dd	221,116,XX
LD	(IX+d),L	DD75dd	221,117,XX
LD	(IX+d),xx	DD36ddxx	221,54,XX,XX
LD	(IY+d),A	FD77dd	253,119,XX
LD	(IY+d),B	FD70dd	253,112,XX
LD	(IY+d),C	FD71dd	253,113,XX
LD	(IY+d),D	FD72dd	253,114,XX
LD	(IY+d),E	FD73dd	253,115,XX
LD	(IY+d),H	FD74dd	253,116,XX

Código de operación		Hex	Decimal
LD	(IY+d),L	FD75dd	253,117,XX
LD	(IY+d),xx	FD36ddxx	253,54,XX,XX
LD	(xxxx),A	32xxxx	50,XX,XX
LD	(xxxx),BC	ED43xxxx	237,67,XX,XX
LD	(xxxx),DE	ED53xxxx	237,83,XX,XX
LD	(xxxx),HL	22xxxx	34,XX,XX
LD	(xxxx),IX	DD22xxxx	221,34,XX,XX
LD	(xxxx),IY	FD22xxxx	253,34,XX,XX
LD	(xxxx),SP	ED73xxxx	237,115,XX,XX
LD	A,(BC)	0A	10
LD	A,(DE)	1A	26
LD	A,(HL)	7E	126
LD	A,(IX+d)	DD7Edd	221,126,XX
LD	A,(IY+d)	FD7Edd	253,126,XX
LD	A,(xxxx)	3Axxxx	58,XX,XX
LD	A,A	7F	127
LD	A,B	78	120
LD	A,C	79	121
LD	A,D	7A	122
LD	A,E	7B	123
LD	A,H	7C	124
LD	A,I	ED57	237,87
LD	A,L	7D	125
LD	A,R	ED5F	237,95
LD	A,xx	3Exx	62,XX
LD	B,(HL)	46	70
LD	B,(IX+d)	DD46dd	221,70,XX
LD	B,(IY+d)	FD46dd	253,70,XX
LD	B,A	47	71
LD	B,B	40	64
LD	B,C	41	65
LD	B,D	42	66
LD	B,E	43	67
LD	B,H	44	68
LD	B,L	45	69
LD	B,xx	06xx	6,XX
LD	BC,(xxxx)	ED4Bxxxx	237,75,XX,XX
LD	BC,xxxx	01xxxx	1,XX,XX
LD	C,(HL)	4E	78
LD	C,(IX+d)	DD4Edd	221,78,XX
LD	C,(IY+d)	FD4Edd	253,78,XX
LD	C,A	4F	79
LD	C,B	48	72
LD	C,C	49	73
LD	C,D	4A	74

Código de operación		Hex	Decimal
LD	C,E	4B	75
LD	C,H	4C	76
LD	C,L	4D	77
LD	C,xx	0Exx	14,XX
LD	D,(HL)	56	86
LD	D,(IX+d)	DD56dd	221,86,XX
LD	D,(IY+d)	FD56dd	253,86,XX
LD	D,A	57	87
LD	D,B	50	80
LD	D,C	51	81
LD	D,D	52	82
LD	D,E	53	83
LD	D,H	54	84
LD	D,L	55	85
LD	D,xx	16xx	22,XX
LD	DE,(xxxx)	ED5Bxxxx	237,91,XX,XX
LD	DE,xxxx	11xxxx	17,XX,XX
LD	E,(HL)	5E	94
LD	E,(IX+d)	DD5Edd	221,94,XX
LD	E,(IY+d)	FD5Edd	253,94,XX
LD	E,A	5F	95
LD	E,B	58	88
LD	E,C	59	89
LD	E,D	5A	90
LD	E,E	5B	91
LD	E,H	5C	92
LD	E,L	5D	93
LD	E,xx	1Exx	30,XX
LD	H,(HL)	66	102
LD	H,(IX+d)	DD66dd	221,102,XX
LD	H,(IY+d)	FD66dd	253,102,XX
LD	H,A	67	103
LD	H,B	60	96
LD	H,C	61	97
LD	H,D	62	98
LD	H,E	63	99
LD	H,H	64	100
LD	H,L	65	101
LD	H,xx	26xx	38,XX
LD	HL,xxxx	21xxxx	33,XX,XX
LD	I,A	ED47	237,71
LD	IX,(xxxx)	DD2Axxxx	221,42,XX,XX
LD	IX,xxxx	DD21xxxx	221,33,XX,XX
LD	IY,(xxxx)	FD2Axxxx	253,42,XX,XX
LD	IY,xxxx	FD21xxxx	253,33,XX,XX

Código de operación		Hex	Decimal
LD	L,(HL)	6E	110
LD	L,(IX+d)	DD6Edd	221,110,XX
LD	L,(IY+d)	FD6Edd	253,110,XX
LD	L,A	6F	111
LD	L,B	68	104
LD	L,C	69	105
LD	L,D	6A	106
LD	L,E	6B	107
LD	L,H	6C	108
LD	L,L	6D	109
LD	L,xx	2Exx	46,XX
LD	R,A	ED4F	237,79
LD	SP,(xxxx)	ED7Bxxxx	237,123,XX,XX
LD	SP,HL	F9	249
LD	SP,IX	DDF9	221,249
LD	SP,IY	FDF9	253,249
LD	SP,xxxx	31xxxx	49,XX,XX
LDD		EDA8	237,168
LDDR		EDB8	237,184
LDI		EDA0	237,160
LDIR		EDB0	237,176
NEG		ED44	237,68
NOP		00	0
OR	(HL)	B6	182
OR	(IX+d)	DDB6dd	221,182,XX
OR	(IY+d)	FDB6dd	253,182,XX
OR	A	B7	183
OR	B	B0	176
OR	C	B1	177
OR	D	B2	178
OR	E	B3	179
OR	H	B4	180
OR	L	B5	181
OR	xx	F6xx	246,XX
OTDR		EDBB	237,187
OTIR		EDB3	237,179
OUT	(C),A	ED79	237,121
OUT	(C),B	ED41	237,65
OUT	(C),C	ED49	237,73
OUT	(C),D	ED51	237,81
OUT	(C),E	ED59	237,89
OUT	(C),H	ED61	237,97
OUT	(C),L	ED69	237,105
OUT	(xx),A	D3xx	211,XX
OUTD		EDAB	237,171

Código de operación		Hex	Decimal
OUTI		EDA3	237,163
POP	AF	F1	241
POP	BC	C1	193
POP	DE	D1	209
POP	HL	E1	225
POP	IX	DDE1	221,225
POP	IY	FDE1	253,225
PUSH	AF	F5	245
PUSH	BC	C5	197
PUSH	DE	D5	213
PUSH	HL	E5	229
PUSH	IX	DDE5	221,229
PUSH	IY	FDE5	253,229
RES	0,(HL)	CB86	203,134
RES	0,(IX+d)	DDCBdd86	221,203,XX,134
RES	0,(IY+d)	FDCBdd86	253,203,XX,134
RES	0,A	CB87	203,135
RES	0,B	CB80	203,128
RES	0,C	CB81	203,129
RES	0,D	CB82	203,130
RES	0,E	CB83	203,131
RES	0,H	CB84	203,132
RES	0,L	CB85	203,133
RES	1,(HL)	CB8E	203,142
RES	1,(IX+d)	DDCBdd8E	221,203,XX,142
RES	1,(IY+d)	FDCBdd8E	253,203,XX,142
RES	1,A	CB8F	203,143
RES	1,B	CB88	203,136
RES	1,C	CB89	203,137
RES	1,D	CB8A	203,138
RES	1,E	CB8B	203,139
RES	1,H	CB8C	203,140
RES	1,L	CB8D	203,141
RES	2,(HL)	CB96	203,150
RES	2,(IX+d)	DDCBdd96	221,203,XX,150
RES	2,(IY+d)	FDCBdd96	253,203,XX,150
RES	2,A	CB97	203,151
RES	2,B	CB90	203,144
RES	2,C	CB91	203,145
RES	2,D	CB92	203,146
RES	2,E	CB93	203,147
RES	2,H	CB94	203,148
RES	2,L	CB95	203,149
RES	3,(HL)	CB9E	203,158
RES	3,(IX+d)	DDCBdd9E	221,203,XX,158
RES	3,(IY+d)	FDCBdd9E	253,203,XX,158

Código de operación		Hex	Decimal
RES	3,A	CB9F	203,159
RES	3,B	CB98	203,152
RES	3,C	CB99	203,153
RES	3,D	CB9A	203,154
RES	3,E	CB9B	203,155
RES	3,H	CB9C	203,156
RES	3,L	CB9D	203,157
RES	4,(HL)	CBA6	203,166
RES	4,(IX+d)	DDCBddA6	221,203,XX,166
RES	4,(IY+d)	FDCBddA6	253,203,XX,166
RES	4,A	CBA7	203,167
RES	4,B	CBA0	203,160
RES	4,C	CBA1	203,161
RES	4,D	CBA2	203,162
RES	4,E	CBA3	203,163
RES	4,H	CBA4	203,164
RES	4,L	CBA5	203,165
RES	5,(HL)	CBAE	203,174
RES	5,(IX+d)	DDCBddAE	221,203,XX,174
RES	5,(IY+d)	FDCBddAE	253,203,XX,174
RES	5,A	CBAF	203,175
RES	5,B	CBA8	203,168
RES	5,C	CBA9	203,169
RES	5,D	CBAA	203,170
RES	5,E	CBAB	203,171
RES	5,H	CBAC	203,172
RES	5,L	CBAD	203,173
RES	6,(HL)	CBB6	203,182
RES	6,(IX+d)	DDCBddB6	221,203,XX,182
RES	6,(IY+d)	FDCBddB6	253,203,XX,182
RES	6,A	CBB7	203,183
RES	6,B	CBB0	203,176
RES	6,C	CBB1	203,177
RES	6,D	CBB2	203,178
RES	6,E	CBB3	203,179
RES	6,H	CBB4	203,180
RES	6,L	CBB5	203,181
RES	7,(HL)	CBBE	203,190
RES	7,(IX+d)	DDCBddBE	221,203,XX,190
RES	7,(IY+d)	FDCBddBE	253,203,XX,190
RES	7,A	CBBF	203,191
RES	7,B	CBB8	203,184
RES	7,C	CBB9	203,185
RES	7,D	CBBA	203,186
RES	7,E	CBBB	203,187

Código de operación		Hex	Decimal
RES	7,H	CBBC	203,188
RES	7,L	CBBD	203,189
RET		C9	201
RET	C	D8	216
RET	M	F8	248
RET	NC	D0	208
RET	NZ	C0	192
RET	P	F0	240
RET	PE	E8	232
RET	PO	E0	224
RET	Z	C8	200
RETI		ED4D	237,77
RETN		ED45	237,69
RL	(HL)	CB16	203,22
RL	(IX+d)	DDCBddl6	221,203,XX,22
RL	(IY+d)	FDCBddl6	253,203,XX,22
RL	A	CB17	203,23
RL	B	CB10	203,16
RL	C	CB11	203,17
RL	D	CB12	203,18
RL	E	CB13	203,19
RL	H	CB14	203,20
RL	L	CB15	203,21
RLA		17	23
RLC	(HL)	CB06	203,6
RLC	(IX+d)	DDCBdd06	221,203,XX,6
RLC	(IY+d)	FDCBdd06	253,203,XX,6
RLC	A	CB07	203,7
RLC	B	CB00	203,0
RLC	C	CB01	203,1
RLC	D	CB02	203,2
RLC	E	CB03	203,3
RLC	H	CB04	203,4
RLC	L	CB05	203,5
RLCA		07	7
RLD		ED6F	237,111
RR	(HL)	CB1E	203,30
RR	(IX+d)	DDCBddlE	221,203,XX,30
RR	(IY+d)	FDCBddlE	253,203,XX,30
RR	A	CB1F	203,31
RR	B	CB18	203,24
RR	C	CB19	203,25
RR	D	CB1A	203,26
RR	E	CB1B	203,27
RR	H	CB1C	203,28
RR	L	CB1D	203,29

Código de operación		Hex	Decimal
RRA		1F	31
RRC	(HL)	CB0E	203,14
RRC	(IX+d)	DDCBdd0E	221,203,XX,14
RRC	(IY+d)	FDCBdd0E	253,203,XX,14
RRC	A	CB0F	203,15
RRC	B	CB08	203,8
RRC	C	CB09	203,9
RRC	D	CB0A	203,10
RRC	E	CB0B	203,11
RRC	H	CB0C	203,12
RRC	L	CB0D	203,13
RRCA		0F	15
RRD		ED67	237,103
RST	0	C7	199
RST	10h	D7	215
RST	18h	DF	223
RST	20h	E7	231
RST	28h	EF	239
RST	30h	F7	247
RST	38h	FF	255
RST	8	CF	207
SBC	A,(HL)	9E	158
SBC	A,(IX+d)	DD9Edd	221,158,XX
SBC	A,(IY+d)	FD9Edd	253,158,XX
SBC	A,A	9F	159
SBC	A,B	98	152
SBC	A,C	99	153
SBC	A,D	9A	154
SBC	A,E	9B	155
SBC	A,H	9C	156
SBC	A,L	9D	157
SBC	A,xx	DExx	222,XX
SBC	HL,BC	ED42	237,66
SBC	HL,DE	ED52	237,82
SBC	HL,HL	ED62	237,98
SBC	HL,SP	ED72	237,114
SCF		37	55
SET	0,(HL)	CBC6	203,198
SET	0,(IX+d)	DDCBddC6	221,203,XX,198
SET	0,(IY+d)	FDCBddC6	253,203,XX,198
SET	0,A	CBC7	203,199
SET	0,B	CBC0	203,192
SET	0,C	CBC1	203,193
SET	0,D	CBC2	203,194
SET	0,E	CBC3	203,195
SET	0,H	CBC4	203,196

Código de operación	Hex	Decimal
SET 0,L	CBC5	203,197
SET 1,(HL)	CBCE	203,206
SET 1,(IX+d)	DDCBddCE	221,203,XX,206
SET 1,(IY+d)	FDCBddCE	253,203,XX,206
SET 1,A	CBCF	203,207
SET 1,B	CBC8	203,200
SET 1,C	CBC9	203,201
SET 1,D	CBCA	203,202
SET 1,E	CBCB	203,203
SET 1,H	CBCC	203,204
SET 1,L	CBCD	203,205
SET 2,(HL)	CBD6	203,214
SET 2,(IX+d)	DDCBddD6	221,203,XX,214
SET 2,(IY+d)	FDCBddD6	253,203,XX,214
SET 2,A	CBD7	203,215
SET 2,B	CBD0	203,208
SET 2,C	CBD1	203,209
SET 2,D	CBD2	203,210
SET 2,E	CBD3	203,211
SET 2,H	CBD4	203,212
SET 2,L	CBD5	203,213
SET 3,(HL)	CBDE	203,222
SET 3,(IX+d)	DDCBddDE	221,203,XX,222
SET 3,(IY+d)	FDCBddDE	253,203,XX,222
SET 3,A	CBDF	203,223
SET 3,B	CBD8	203,216
SET 3,C	CBD9	203,217
SET 3,D	CBDA	203,218
SET 3,E	CBDB	203,219
SET 3,H	CBDC	203,220
SET 3,L	CBDD	203,221
SET 4,(HL)	CBE6	203,230
SET 4,(IX+d)	DDCBddE6	221,203,XX,230
SET 4,(IY+d)	FDCBddE6	253,203,XX,230
SET 4,A	CBE7	203,231
SET 4,B	CBE0	203,224
SET 4,C	CBE1	203,225
SET 4,D	CBE2	203,226
SET 4,E	CBE3	203,227
SET 4,H	CBE4	203,228
SET 4,L	CBE5	203,229
SET 5,(HL)	CBEE	203,238
SET 5,(IX+d)	DDCBddEE	221,203,XX,238
SET 5,(IY+d)	FDCBddEE	253,203,XX,238
SET 5,A	CBEF	203,239
SET 5,B	CBE8	203,232

Código de operación	Hex	Decimal
SET 5,C	CBE9	203,233
SET 5,D	CBEA	203,234
SET 5,E	CBEB	203,235
SET 5,H	CBEC	203,236
SET 5,L	CBED	203,237
SET 6,(HL)	CBF6	203,246
SET 6,(IX+d)	DDCBddF6	221,203,XX,246
SET 6,(IY+d)	FDCBddF6	253,203,XX,246
SET 6,A	CBF7	203,247
SET 6,B	CBF0	203,240
SET 6,C	CBF1	203,241
SET 6,D	CBF2	203,242
SET 6,E	CBF3	203,243
SET 6,H	CBF4	203,244
SET 6,L	CBF5	203,245
SET 7,(HL)	CBFE	203,254
SET 7,(IX+d)	DDCBffFE	221,203,XX,254
SET 7,(IY+d)	FDCBddFE	253,203,XX,254
SET 7,A	CBFF	203,255
SET 7,B	CBF8	203,248
SET 7,C	CBF9	203,249
SET 7,D	CBFA	203,250
SET 7,E	CBFB	203,251
SET 7,H	CBFC	203,252
SET 7,L	CBFD	203,253
SLA (HL)	CB26	203,38
SLA (IX+d)	DDCBdd26	221,203,XX,38
SLA (IY+d)	FDCBdd26	253,203,XX,38
SLA A	CB27	203,39
SLA B	CB20	203,32
SLA C	CB21	203,33
SLA D	CB22	203,34
SLA E	CB23	203,35
SLA H	CB24	203,36
SLA L	CB25	203,37
SRA (HL)	CB2E	203,46
SRA (IX+d)	DDCBdd2E	221,203,XX,46
SRA (IY+d)	FDCBdd2E	253,203,XX,46
SRA A	CB2F	203,47
SRA B	CB28	203,40
SRA C	CB29	203,41
SRA D	CB2A	203,42
SRA E	CB2B	203,43
SRA H	CB2C	203,44
SRA L	CB2D	203,45

Código de operación		Hex	Decimal
SRL	(HL)	CB3E	203,62
SRL	(IX+d)	DDCBdd3E	221,203,XX,62
SRL	(IY+d)	FDCBdd3E	253,203,XX,62
SRL	A	CB3F	203,63
SRL	B	CB38	203,56
SRL	C	CB39	203,57
SRL	D	CB3A	203,58
SRL	E	CB3B	203,59
SRL	H	CB3C	203,60
SRL	L	CB3D	203,61
SUB	(HL)	96	150
SUB	(IX+d)	DD96dd	221,150,XX
SUB	(IY+d)	FD96dd	253,150,XX
SUB	A	97	151
SUB	B	90	144
SUB	C	91	145
SUB	D	92	146
SUB	E	93	147
SUB	H	94	148
SUB	L	95	149
SUB	xx	D6xx	214,XX
XOR	(HL)	AE	174
XOR	(IX+d)	DDAEdd	221,174,XX
XOR	(IY+d)	FDAEdd	253,174,XX
XOR	A	AF	175
XOR	B	A8	168
XOR	C	A9	169
XOR	D	AA	170
XOR	E	AB	171
XOR	H	AC	172
XOR	L	AD	173
XOR	xx	EExx	238,XX

APENDICE C

N. de la Ed. — Para facilitar la comprensión de las instrucciones del Código Máquina que se presentan a continuación, se incluyen unas explicaciones previas que pueden resultar orientativas.

S — Este señalizador refleja el valor del bit más significativo del byte sobre el que se esté trabajando (bit 7). En la notación de complemento a 2, este bit indica el signo del número. El 0 indica un número positivo y el 1 un número negativo. Por esta razón, el bit 7 se denomina «bit de signo».

Z — Este señalizador indica que el byte con el que se está trabajando tiene valor 0. Cuando esto ocurre, el señalizador Z toma valor 1. En caso de que el resultado no sea 0, el bit Z tiene valor 0.

—El guión indica que el bit no ha sido afectado por la operación realizada.

H — Este es el señalizador de «medio acarreo» (half carry). Este señalizador indica que, tras una operación aritmética, se ha producido un acarreo del bit 3 sobre el 4. Dicho de otra forma, este señalizador indica la exigencia de un acarreo de los cuatro bits menos significativos sobre los cuatro bits más significativos. Se utiliza en operaciones en aritmética DCB (Decimal Coded Binary = Decimal Codificado en Binario).

P — Este es el señalizador de paridad. Algunas instrucciones cambian el valor de este bit según la paridad del resultado. Esta se determina contando el número de unos que hay en el resultado.

Si el número es impar el señalizador de paridad se pone a cero (lo que indica paridad impar). Si es par, el señalizador se pone a 1 (lo que indica paridad par). La utilización más frecuente de la paridad se relaciona con la gestión de bloques de caracteres (generalmente en ASCII). El bit de paridad se añade a los siete bits con los que se codifica el carácter con el objeto de comprobar la validez de los datos almacenados en memoria. Por ejemplo, si accidentalmente se alterara uno de los bits de los siete que se utilizan para representar un carácter (este accidente puede ocurrir durante la transmisión de datos o por mal funcionamiento de la memoria), entonces el número de unos dentro de estos siete bits cambiará. Comprobando el valor del bit de paridad se puede detectar este cambio y actuar en consecuencia.

N — Este es el señalizador de substracción. No es normalmente utilizado por el programador. Es el Z80 el que, de una manera interna, lo emplea en operaciones DCB. Después de una operación de substracción o adición en DCB, es necesario realizar una operación DAA (Ajuste Decimal del Acumulador) para obtener el resultado correcto. Esta operación de ajuste no es la misma para la suma que para la substracción. Según el valor del señalizador N, se ejecuta una u otra. En el caso de que el valor sea 1, el señalizador se pone a 1; en el caso de que el valor sea 0, el señalizador se pone a 0.

C — Este señalizador tiene una función doble: indicar que se ha producido un acarreo durante una operación de substracción o de suma y actuar como noveno bit para almacenamiento temporal en las operaciones de desplazamiento.

@ — Este signo indica que el bit ha sido modificado como resultado de una operación, en la columna en la que aparece este signo.

? — El interrogante indica que el señalizador queda afectado aleatoriamente por el resultado de la operación.

INSTRUCCIONES	SEÑALIZADORES							
Código de operación	S	Z	–	H	–	P	N	C
ADC A,r	@	@	–	@	–	@	0	@
ADC HL,s	@	@	–	@	–	@	0	@
ADD A,r	@	@	–	@	–	@	0	@
ADD HL,s	–	–	–	@	–	–	0	@
ADD IX,s	–	–	–	@	–	–	0	@
ADD IY,s	–	–	–	@	–	–	0	@
AND r	@	@	–	1	–	@	0	0
BIT b,r	?	@	–	1	–	@	0	0
CALL pq	–	–	–	–	–	–	–	–
CALL c,pq	–	–	–	–	–	–	–	–
CCF	–	–	–	x	–	–	0	@
(el señalizador H toma el valor previo del señalizador C)								
CP r	@	@	–	@	–	@	1	@
CPI	@	x	–	@	–	x	1	–
CPD	@	x	–	@	–	x	1	–
CPIR	@	x	–	@	–	x	1	–
CPDR	@	x	–	@	–	x	1	–
(Z) toma valor 1 si BC toma valor cero. P/V se convierte en 1 si A = (HL-1)).								
CPL	–	–	–	1	–	–	1	–
DAA	@	@	–	@	–	@	–	@
DEC r	@	@	–	@	–	@	1	–
DEC s	–	–	–	–	–	–	–	–
DI	–	–	–	–	–	–	–	–
DJNZ e	–	–	–	–	–	–	–	–
EI	–	–	–	–	–	–	–	–
EX AF, AF'	–	–	–	–	–	–	–	–
EX DE,HL	–	–	–	–	–	–	–	–
EX (SP),HL	–	–	–	–	–	–	–	–
EX (SP),IX	–	–	–	–	–	–	–	–
EX (SP),IY	–	–	–	–	–	–	–	–
EXX	–	–	–	–	–	–	–	–
HALT	–	–	–	–	–	–	–	–
IM 0	–	–	–	–	–	–	–	–
IM 1	–	–	–	–	–	–	–	–
IM 2	–	–	–	–	–	–	–	–
INC r	@	@	–	@	–	@	0	–
INC s	–	–	–	–	–	–	–	–
IN A,(n)	–	–	–	–	–	–	–	–
IN r,(C)	@	@	–	@	–	@	0	–
INI	?	x	–	?	–	?	1	–
IND	?	x	–	?	–	?	1	–
(Z toma valor 1 si B toma valor cero).								
INIR	?	1	–	?	–	?	1	–
INDR	?	1	–	?	–	?	1	–
JP pq	–	–	–	–	–	–	–	–
JP c,pq	–	–	–	–	–	–	–	–
JP (HL)	–	–	–	–	–	–	–	–
JP (IX)	–	–	–	–	–	–	–	–
JP (IY)	–	–	–	–	–	–	–	–
JR e	–	–	–	–	–	–	–	–
JR c,e	–	–	–	–	–	–	–	–
LD(BC),A	–	–	–	–	–	–	–	–
LD A,(BC)	–	–	–	–	–	–	–	–
LD (DE),A	–	–	–	–	–	–	–	–
LD A,(DE)	–	–	–	–	–	–	–	–
LD I,A	–	–	–	–	–	–	–	–

INSTRUCCIONES	SEÑALIZADORES							
Código de operación	S	Z	–	H	–	P	N	C
LD R,A	–	–	–	–	–	–	–	–
LD A,I	@	@	–	0	–	x	0	–
LD A,R	@	@	–	0	x	0	–	
(Cuando P/V se activa para interrumpir el señalizador de almacenamiento).								
LD SP,HL	–	–	–	–	–	–	–	–
LD SP,IX	–	–	–	–	–	–	–	–
LD SP,IY	–	–	–	–	–	–	–	–
LD r,r	–	–	–	–	–	–	–	–
LD s,mn	–	–	–	–	–	–	–	–
LD A,(pq)	–	–	–	–	–	–	–	–
LD s,(pq)	–	–	–	–	–	–	–	–
LD (pq),A	–	–	–	–	–	–	–	–
LD (pq),s	–	–	–	–	–	–	–	–
LDI	–	–	–	0	–	x	0	–
LDD	–	–	–	0	–	x	0	–
(P/V se convierte en 0 se BC se convierte en 0).								
LDIR	–	–	–	0	–	0	0	–
LDDR	–	–	–	0	–	0	0	–
NEG	@	@	–	@	–	@	1	@
NOP	–	–	–	–	–	–	–	–
OR r	@	@	–	0	–	@	0	0
OUT (n),A	–	–	–	–	–	–	–	–
OUT (C),r	–	–	–	–	–	–	–	–
OUTI	?	x	–	?	–	?	1	–
OUTD	?	x	–	?	–	?	1	–
(Z toma el valor 1 si B toma el valor cero).								
OTIR	?	1	–	?	–	?	1	–
OTDR	?	1	–	?	–	?	1	–
POP AF	x	x	x	x	x	x	x	x
(El valor de los señalizadores queda determinado por el byte que ocupa la parte superior de la pila).								
POP s	–	–	–	–	–	–	–	–
PUSH AF	–	–	–	–	–	–	–	–
PUSH s	–	–	–	–	–	–	–	–
RES b,r	–	–	–	–	–	–	–	–
RET	–	–	–	–	–	–	–	–
RET c	–	–	–	–	–	–	–	–
RETN	–	–	–	–	–	–	–	–
RETI	–	–	–	–	–	–	–	–
RLA	–	–	–	0	–	–	0	@
RL r	@	@	–	0	–	@	0	@
RLCA	–	–	–	–	–	–	–	–
RES b,r	–	–	–	–	–	–	–	–
RET	–	–	–	–	–	–	–	–
RET c	–	–	–	–	–	–	–	–
RETN	–	–	–	–	–	–	–	–
RETI	–	–	–	–	–	–	–	–
RLCA	–	–	–	0	–	–	0	@
RRCA	–	–	–	0	–	–	0	@
RLA	–	–	–	0	–	–	0	@
RRA	–	–	–	0	–	–	0	@
RLC r	@	@	–	0	–	@	0	@
RRC r	@	@	–	0	–	@	0	@
RL r	@	@	–	0	–	@	0	@
RR r	@	@	–	0	–	@	0	@
RRD	@	@	–	0	–	@	0	–
RLD	@	@	–	0	–	@	0	–
RST 00	–	–	–	–	–	–	–	–
RST 08	–	–	–	–	–	–	–	–
RST 10	–	–	–	–	–	–	–	–
RST 18	–	–	–	–	–	–	–	–
RST 20	–	–	–	–	–	–	–	–

INSTRUCCIONES	SEÑALIZADORES							
Código de operación	**S**	**Z**	**–**	**H**	**–**	**P**	**N**	**C**
RST 28	–	–	–	–	–	–	–	–
RST 30	–	–	–	–	–	–	–	–
RST 38	–	–	–	–	–	–	–	–
SBC A,r	@	@	–	@	–	@	1	@
SBC HL,s	@	@	–	@	–	@	1	@
SCF	–	–	–	0	–	–	0	1
SET b,r	–	–	–	–	–	–	–	–
SLA r	@	@	–	0	–	@	0	@
SRA r	@	@	–	0	–	@	0	@
SLR r	@	@	–	0	–	@	0	@
SUB r	@	@	–	@	–	@	1	@
XOR r	@	@	–	0	–	@	0	0

GUÍA PRÁCTICA DE BASIC del ZX-81 y del SPECTRUM

por **RAMÓN ROVIRA SOLIGÓ**

Libro n.° 172

148 págs.
31 figuras

SUMARIO

Introducción a los ordenadores - Instalación y mantenimiento - La familiarización con el ZX-81 - Empezando a programar - Instrucciones principales - Operaciones y funciones - El juego de caracteres - Las cadenas - Condiciones y operadores lógicos - Programación con bucles - Conjuntos, vectores y matrices - Gráficos - La impresora - El almacenamiento de datos en cassette - Estructura de la memoria - Acceso a la memoria - PEEK y POKE - Variables del sistema - Instrucción a código máquina - Ahorro de memoria - Comparación con otros BASICs - Códigos de error - El ZX-Spectrum - Teclado - Características adicionales del ZX-Spectrum sobre el ZX-81 - Color; sonido; separación de instrucciones en una misma línea - Instrucciones y comandos BASIC - Gráficos de alta resolución - Almacenamiento de programas y datos en cassette - Juego de caracteres - Mensajes de error - Microdrives - Mapa de memoria

ACCESO RÁPIDO AL VIC-20

por **TIM HARTNELL**

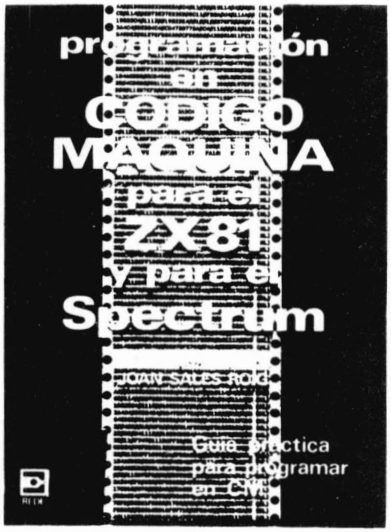
Libro n.° 173

148 págs.

LISTADOS EN ESPAÑOL DE TODOS LOS PROGRAMAS

SUMARIO

Números aleatorios - Mecanismos simples de toma de decisión - La ruleta rusa - El "núcleo de programa" , la premisa para buenos juegos - Realización de gráficos - Cómo se forma la biblioteca de programas propia - For/Next - Cara o cruz - Cálculos numéricos - Ideas generales sobre la confección de programas - Percepción extrasensorial - Mata al asesino - ¿Cuánto tiempo me queda? - Jugando con las musas - Búsqueda de coordenadas - Tragaperras - Perdidos en el espacio - Vectores ("arrays") - La araña dimensional - "Pesky piksy" - Alienígenas y asteroides - Funciones gráficas en color - Serpientes y escaleras - Carreras de coches - Juego de cartas - Acertijo - El juego de las damas - Primeros pasos hacia las estrellas - Frenesí - Posibilidades musicales del VIC-20 - Instrucciones READ y DATA - Maestro - Escalas de dificultad - La cámara de ecos - Qué más puede hacer con su VIC-20 - El rey de las cavernas - Alfa - Cambio de papeles - El colgado (las nueve vidas) - Nim - El juego de la vida - Cubik - ¿Cuál es el número elegido? - Lógica - "Blackjack" - Kalki, el adivinador del pensamiento - "Chemin de fer" - "Craps" - "Esperanza de vida" - "La pajarera" - El séptimo cielo - Alunizaje - Laberinto - El profesor - La conversión de temperaturas - El acertijo de la multiplicación - Cuadrados - Apéndices: Comandos; Instrucciones; Funciones; Código de pantalla y A$



PROGRAMACIÓN EN CÓDIGO MÁQUINA PARA EL ZX-81 Y PARA EL SPECTRUM

por **JOAN SALES ROIG**

Libro n.° 175

158 págs.
10 figuras

SUMARIO

Introducción - Decimal, Hexadecimal, Binario - BASIC y Código Máquina - Los registros - Código Máquina en el ZX-81 - La instrucción LD - Aritmética sencilla: ADD, SUB, INC, DEC - Los señalizadores (flags) - Instrucciones relacionadas con flags - Saltos: absolutos, relativos y condicionales - La pila ("stack") - Trabajando con bits - Bucles y subrutinas - Instrucciones con repetición - Juego completo de instrucciones del Z-80 - Estructura de la pantalla: D- ILE - El teclado - Conexiones MIC y EAR - Técnicas de programación - Aplicación práctica: juego "Comecocos" - Direcciones útiles de la ROM - Cálculos más complejos con la ROM - Instrucciones del Z-80 - Tabla de alteraciones de flags - Tablas de saltos relativos hacia adelante y hacia atrás - Mapa de memoria - La estructura de pantalla en el ZX Spectrum - El teclado en el ZX Spectrum - El altavoz y las conexiones MIC y EAR en el Spectrum - Consideraciones para aplicar el contenido del libro al ZX Spectrum - Modificaciones en el juego "Comecocos"

LA MEJOR PROGRAMACIÓN DEL ZX-SPECTRUM POR LA PRÁCTICA

por **TIM HARTNELL** y **DILWYN JONES**

Libro n.° 177
244 págs.

LISTADOS EN ESPAÑOL DE TODOS LOS PROGRAMAS

SUMARIO

Utilización del teclado - PRINT y TAB - Conservación de programas (SAVE) - Comprobación de la buena grabación de un programa - Combinación de dos programas en el ordenador - Consejos para resolver en la carga de programas - PRINT AT - Colores y gráficos - Gráficos con gran resolución - Trazado de dibujos - CIRCLE - El arte de utilizar las cadenas - Figuras de moiré - POINT - La impresora - Números aleatorios - Corrida de toros - Variables - Variables en cadena - El canto de los grillos y la temperatura - INPUT - Combate - Interés compuesto - GO TO - IF... THEN GO TO - IF/THEN/ELSE - Bucles FOR/NEXT - Bucles anidados - STEP - GOSUB y RETURN - Sonido - Definición de funciones - La sentencia DIM y los conjuntos - Rompecódigos - Conjuntos de cadenas o variables alfanuméricas - Manejo de cadenas - Las funciones LEN y STR$ - INKEY$ - READ/DATA/STORE - Gráficos definidos por el usuario - DOTMAN (comecocos) - Eliminación de una parte de la presentación en pantalla - Desplazamiento de pantalla en BASIC (Scrolling) - Desplazamiento ascendente y descendente, hacia la izquierda y hacia la derecha - Conservación de líneas en los márgenes - Movimiento de gráficos - SCREEN$ - La flecha roja - Engullidor de basura - Manipulación de cadenas alfanuméricas - INKEY$ cambiado - Introducción a la aritmética - Números primos - Estadística - Evolución de dos especies - Funciones trigonométricas - Conversión de otros dialectos del BASIC - Aritmética de números enteros - Funciones GET y GET$, VAL, SET, RESET, ELSE, REPEAT... - UNTIL - Variables no definidas - Matrices - Grados y radianes - Logaritmos base 1 [1] - Porcentaje - Interrrogación - PEEK y POKE - Aplicaciones comerciales - Tratamiento de textos - Cómo mejorar los programas - Color - "Vida" - Juego de las cerillas - Circuito final - Estallido - Galaxian - Apéndices

ediciones
técnicas
REDE
BARCELONA
(ESPAÑA)